

AD-A248 007



2



Ada

DTIC
ELECTE
MAR 18 1992
S C D

PROCEEDINGS OF THE TENTH ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

FEBRUARY 24-28, 1992

Sponsored By:
ANCOST, INC.

Approved for public release;
Distribution Unlimited

With Participation By:
UNITED STATES ARMY
UNITED STATES NAVY
UNITED STATES AIR FORCE
UNITED STATES MARINE CORPS
FEDERAL AVIATION ADMINISTRATION
DEFENSE INFORMATION SYSTEMS AGENCY
ADA JOINT PROGRAM OFFICE
NATIONAL AERONAUTICS & SPACE ADMINISTRATION

Academic Host:
MOREHOUSE COLLEGE

92-06791



92 3 16 087

PROCEEDINGS OF TENTH ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

Sponsored By:
ANCOST, INC.



With Participation By:
UNITED STATES ARMY
UNITED STATES NAVY
UNITED STATES MARINE CORPS
UNITED STATES AIR FORCE
FEDERAL AVIATION ADMINISTRATION
DEFENSE INFORMATION SYSTEMS AGENCY
ADA JOINT PROGRAM OFFICE
NATIONAL AERONAUTICS & SPACE ADMINISTRATION

Academic Host:
MOREHOUSE COLLEGE

HYATT REGENCY-CRYSTAL CITY, ARLINGTON, VA

February 24-28, 1992

Accession For	
DTIC GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A1	

10th ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

CONFERENCE COMMITTEE 1990-1991

Executive Committee Chair:
MS. DEE M. GRAUMANN
 General Dynamics, ED
 San Diego, CA 92186-5227

Treasurer:
MS. SUSAN MARKEL
 TRW
 Fairfax, VA 22031

Secretary:
MR. STEVE LAZAROWICH
 Alsys
 Reston, VA 22090

Immediate Past-Chair:
DR. M. SUSAN RICHMAN
 The Pennsylvania University at
 Harrisburg
 Middletown, PA 17057

Conference Chair:
MS. JUDITH M. GILES
 Intermetrics
 Cambridge, MA 02138

Academic Outreach:
DR. ART JONES
 Morehouse College
 Atlanta, GA 30314

Budget Committee Chair:
MR. DONALD C. FUHR
 Tuskegee University
 Tuskegee, AL 36088

**Policies Procedures & By-Laws
 Committee**
Chair:
DR. RICHARD KUNTZ
 Monmouth College
 W Long Branch, NJ 07764

Public Relations Chair:
MS. LAURA VEITH
 Rational
 Bethesda, MD 20817

Local Arrangements Chair:
MR. STEVE LAZEROWICH
 Alsys
 Reston, VA 22090

Exhibition Co-Chair:
MR. MICHAEL SAPENTER
 Telos Systems Group
 Lawton, OK 73501

Exhibition Co-Chair:
MR. GLENN HUGHES
 Rational
 Bethesda, MD 20817-1007

Panels Chair:
DR. CHARLES LILLIE
 SAIC
 McLean, VA 22102

Tutorials Chair:
DR. M. SUSAN RICHMAN
 The Pennsylvania University at
 Harrisburg
 Middletown, PA 17057

Technical Program Chair:
MR. DANIEL E. HOCKING
 AIRMICS
 Atlanta, GA 30332-0800

MS. CHRISTINE L. BRAUN
 GTE Federal Systems
 Chantilly, VA 22021-3808

MR. MIGUEL A. CARRIO, JR.
 MTM Engineering
 McLean, VA 22102-8501

MS. LUANA CLEVER
 Florida Institute of Technology
 Melbourne, FL 32901

DR. VERLYNDA DOBBS
 Telos Corporation
 Shrewsbury, NJ 07702

DR. GENEVIEVE M. KNIGHT
 Coppin State College
 Baltimore, MD 21216

CONFERENCE DIRECTOR:
MARJORIE Y. RISINGER, CMP
 Rosenberg & Risinger, Inc.
 Culver City, CA 90230

ADVISORY MEMBERS

Maj. TOM CROAK
 US Air Force
 Washington, DC 20330-5190

MR. JACK DANIELS
 USAISSC
 Ft. Belvoir, VA 22060-5459

Capt. HUET LANDRY
 US Air Force/DISA
 Fort Monmouth, NJ 07703-5513

DR. JOHN SOLOMOND
 AJO
 Washington, DC 20310

MR. CARRINGTON STEWART
 NASA
 Houston, TX 77058

MS. ANTOINETTE STUART
 US Navy
 Washington, DC 20734-5072

MAJ. DAVID THOMPSON
 US Marine Corps
 Washington, DC 20380-0001

MS. KAY TREZZA
 HQ, CECOM, CSE
 Ft. Monmouth, NJ 07703-5000

DR. GEORGE W. WATTS
 HQ, CECOM, CSE
 Ft. Monmouth, NJ 07703-5000

MS. ALICE WONG
 FAA
 Washington, DC 20024

PANELS AND TECHNICAL SESSIONS

Tuesday, February 25, 1992

8:30 AM Opening Session
 10:15 AM Acquisition Panel
 2:00 PM Reuse: Domain Analysis
 2:00 PM Education
 2:00 PM Development Methods
 2:00 PM Object Oriented
 4:00 PM DISA's Roles in the Center of Information Management

Wednesday, February 26, 1992

8:30 AM Opening Session
 9:00 AM CIM Panel
 10:45 AM Reuse: Abstract Data Types
 10:45 AM Management of Software Development
 10:45 AM Development Methods 11
 10:45 AM Student Papers
 2:15 PM Reuse: Reverse Engineering
 2:15 PM Metrics
 2:15 PM Software Process Improvement Panel
 4:00 PM Preparing Students for Industry Panel
 4:00 PM Software Reuse Panel
 7:00 PM Reuse Business Issues Birds of a Feather

Thursday, February 27, 1992

8:30 AM Reuse: Architecture
 8:30 AM Real-Time
 8:30 AM Ada Applications in Aeronautics & Space Systems
 Development at NASA Panel
 10:30 AM Ada 9X Panel
 10:30 AM Reuse: General
 10:30 AM Artificial Intelligence
 2:00 PM Futures Panel

Papers

The papers in this volume were printed directly from unedited reproducible copies prepared by the authors. Responsibility for contents rests upon the author, and not the symposium committee or its members. After the symposium, all publication rights of each paper are reserved by their authors, and requests for republication of a paper should be addressed to the appropriate author. Abstracting is permitted, and it would be appreciated if the symposium is credited when abstracts or papers are republished. Requests for individual copies of papers should be addressed to the authors.

PROCEEDINGS

TENTH ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

Bound-available at Fort Monmouth

2nd Annual National Conference on Ada Technology, 1984—(Not Available)
3rd Annual National Conference on Ada Technology, 1985—\$10.00
4th Annual National Conference on Ada Technology, 1986—(Not Available)
5th Annual National Conference on Ada Technology, 1987—(Not Available)
6th Annual National Conference on Ada Technology, 1988—\$20.00
7th Annual National Conference on Ada Technology, 1989—\$20.00
8th Annual National Conference on Ada Technology, 1990—\$25.00
9th Annual National Conference on Ada Technology, 1991—\$25.00
10th Annual National Conference on Ada Technology, 1992—\$25.00

Extra Copies: 1-3 \$25 each; next 7 \$20 each; 11 & more \$15 each.

Make check or bank draft payable in U.S. dollars to ANCOST and forward requests to:

Annual National Conference on Ada Technology
U.S. Army Communications-Electronics Command
ATTN: AMSEL-RD-SE-CRM (Ms. Kay Trezza)
Fort Monmouth, NJ 07703-5000

Telephone inquiries may be directed to Ms. Kay Trezza at 908/532-1 898.

Photocopies—Available at Department of Commerce. Information on prices and shipping charges should be requested from:

U.S. Department of Commerce
National Technical Information Service
Springfield, VA 22151
USA

Include title, year, and AD number

2nd Annual National Conference on Ada Technology, 1984—AD A142403
3rd Annual National Conference on Ada Technology, 1985—AD A164338
4th Annual National Conference on Ada Technology, 1986—AD A167802
5th Annual National Conference on Ada Technology, 1987—AD A178690
6th Annual National Conference on Ada Technology, 1988—AD A190936
7th Annual National Conference on Ada Technology, 1989—AD A217979
8th Annual National Conference on Ada Technology, 1990—AD A219777
9th Annual National Conference on Ada Technology, 1991—AD A233469

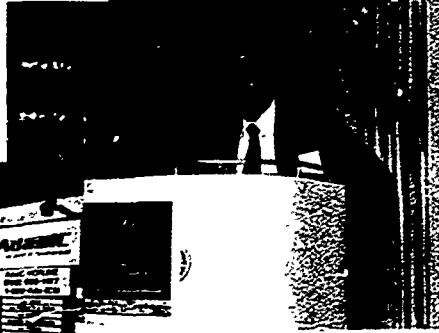
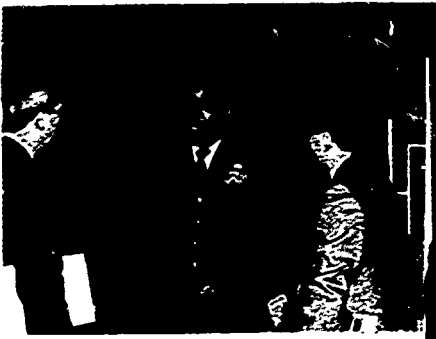
WASHINGTON HILTON & TOWERS WASHINGTON, DC 1991 Ada Conference



Conference Highlights

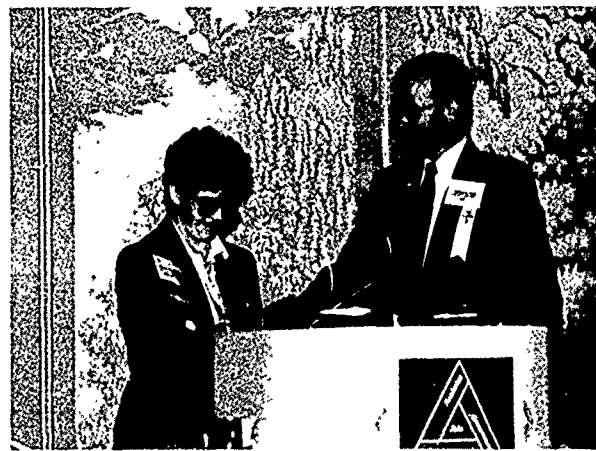






9th ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

Leadership Awards



Thank you

MESSAGE FROM THE CONFERENCE CHAIRPERSON

On behalf of the participating government agencies, the Conference Committee, and Morehouse College, welcome to the 10th Annual National Conference on Ada Technology. Committee members and their supporting organizations are listed in the proceedings and will be recognized during the conference. Morehouse College is this year's host college -- a fitting role since they helped found the conference.



This year's program takes a close look at initiatives and technologies impacting software acquisition and development. Senior officers will discuss current acquisition policies and procedures. Paul Strassmann, the services, and DISA will offer their perspectives on the Corporate Information Management (CIM) initiative. Software reuse, software process improvement, and the current status of Ada9X will be explored. Representatives from Government, industry, and academia will describe frankly what it takes to remain competitive. In addition, there will be technical papers on a wide range of lessons learned and state of technology topics and issues. Finally, an Exhibitors Showcase will give you a chance to see first-hand what is being discussed in the sessions.

Education is vital to the conference. Tutorials are scheduled for managers, implementors, and trainers. A major goal of this conference, however, is to help academic institutions, particularly the Historically Black Colleges and Universities (HBCUs), prepare software engineers and scientists for this nation's work force. You will note many students in attendance and a special session for student papers. Please take the time to introduce yourself to these special guests, discuss your organization and the role you play, and help them any way you can.

In honor of the conference's 10th anniversary, the Founders Award will be presented during the Opening Session. In addition, we welcome back guest speakers whose on-going support typifies the services' commitment to this forum. Finally, throughout the conference, we will pay special tribute to those individuals and organizations who got it started and were instrumental in keeping it going.

The conference's success depends on continued support from many individual companies, government agencies, and participating colleges. The Conference Committee welcomes your comments and suggestions for improvements. Also, we invite you to speak with a committee member or someone at the registration desk if you would like to become more actively involved.

We are pleased you chose to attend and hope you will join us again next year. The 11th Annual Conference on Ada Technology (1993) will be held here at the Hyatt Regency - Crystal City, Arlington, Virginia. The dates are 8-12 March 1993.

Dee Graumann
Dee Graumann

A HISTORY OF SUCCESS

This conference grew out of Equal Employment Opportunities (EEO) initiatives in the United States Army's Center for Tactical Computer Systems (CENTACS) at Ft. Monmouth, New Jersey. In the early 1980s, there simply were not enough technically qualified minority applicants available to meet CENTACS' Affirmative Action performance initiatives. CENTACS looked to academia and industry to help it grow such capability. The goal was to develop a means of educating minority students to become computer scientists and software engineers. Regional institutions that participated in the Army's Summer Professors Program and Historically Black Colleges and Universities (HBCUs) were contacted and encouraged to adapt their curricula to these requirements. Also, industry was encouraged to enhance the schools' capability to respond by donating equipment and tools or providing them at cost. At the same time, the Communications-Electronics Command (CECOM) at Ft. Monmouth was deeply involved with establishing Ada as the Army's standard software development language. These two needs merged in 1982 when CECOM, SofTech, Inc., Morehouse College, and Hampton University agreed to co-sponsor the Southeastern Ada Tutorial Conference in Atlanta, Georgia. The conference focused on training and educating academia on the Ada language and software development requirements. The results heartened all planners. Organizations and institutions other than the HBCUs participated. Industry arrived with exhibits and resources to support the institutions. Other government organizations participated fully in the proceedings. Attendees from the software development community seemed eager for any information they could get to accelerate their own understanding and ability to respond to Ada requirements. An on-going conference was established to promote software technology education. Ada was to be the focus until the language matured or a new software-related technology priority was established. The following table traces the conference's history:

#	YEAR	SITE	ACADEMIC HOST
1	1983	Atlanta, GA	Morehouse College, Hampton University
2	1984	Hampton, VA	Hampton University
3	1985	Houston, TX	Prairie View A&M
4	1986	Atlanta, GA	Atlanta University
5	1987	Arlington, VA	Howard University
6	1988	Arlington, VA	Norfolk State University, University of Maryland
7	1989	Atlantic City, NJ	Monmouth College, Jersey State College, Cheyney University, Penn State/Harrisburg, Stockton State College
8	1990	Atlanta, GA	Georgia Institute of Technology, Morehouse College, Tuskegee University
9	1991	Washington, DC	Coppin State College
10	1992	Arlington, VA	Morehouse College

Today the conference is planned and executed by ANCOST, Inc., a board comprised of thirty-one individuals from volunteering government, academia, and industry organizations. Government participants include the U.S. Army, U.S. Air Force, U.S. Navy, U.S. Marine Corps, DISA, NASA, and FAA. Ada is still an underlying concern, but primarily as an enabling technology. Today's focus is on software acquisition and development in an environment where priorities and expectations have changed enormously since the early 1980s. In addition to maintaining the conference, the board administers a program to involve students in the conference and supply software development tools to academic institutions in need at little or no cost. We thank you for, and encourage, your continuing participation and support in this endeavor.

ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

FOUNDERS DAY RECOGNITION

The Annual National Conference on Ada Technology has grown out of an Ada Education Initiative for Historically Black Colleges and Universities to become a major forum for software acquisition and development issues, policies, and procedures and Ada education. Many people helped make this happen during the past decade. This conference, however, pays special tribute to those who grew the concept and were instrumental in making this annual conference, as well as the special programs for colleges and universities, an on-going reality. Those people, as well as the organizations that committed to sponsoring their participation at the time, are listed below. We will be asking any who are in attendance to stand and be recognized throughout the conference. If you were overlooked, please accept our apologies and do stand and accept our recognition as well.

<u>NAME</u>	<u>ORGANIZATION</u>
Christine Braun*	SofTech, Inc.
Miguel Carrio*	CENTACS
Mike Danko	RCA
Don Fuhr*	Tuskegee University
David Haratz	CENTACS
Charlene Hayden	GTE
Hugh Gloster	Morehouse College
Art Jones *	Morehouse College
Joe Kernan	CENTACS
Genevieve Knight*	Hampton University
Kurth Krause	Intermetrics
Nico Lomuto	SofTech, Inc.
Benjamin Martin	Atlanta University
Edith W. Martin	DOD(USDR&E)
Isabel Muennichow	TRW
Mac Murray	General Dynamics
Emmett Paige, Jr.	U.S. Army
Robert Rechter	TRW
Susan Richman*	Penn State
John Roberts	BDM
Jorge Rodriguez	SofTech, Inc.
Ruth Rudolph	CSC
Alan Salisbury	U.S. Army
Jim Schell	CENTACS
Lawrence Straw	RCA
Ken Taormina	Teledyne
Putnam Texel	SofTech, Inc.
Dennis Turner	CENTACS
Thomas Wheeler	CENTACS
Bronel Whelchel**	CENTACS
Martin Wolfe	CENTACS
Paul Wolfgang	CSC

* Currently members of ANCOST, Inc. board.

** Deceased

TABLE OF CONTENTS

Tuesday, February 25, 1992

Exhibit Hours: 3:30 - 7:00pm

Opening Session: 8:30 - 10:30am

Keynote Speaker: Mr. Grady Booch, Director of Object-Oriented Products, Rational.....2

Acquisition Panel: 10:15am - 12 Noon.....10

Moderator: Mr. John H. Sirtic, Director, Software Engineering, US Army CECOM, Ft. Monmouth, NJ.....3

Panelists: MG Alfred J. Malette, Commanding General, United States Army Communications-Electronics Command, Fort Monmouth, NJ.....4

Mr. Michael Perie, Advanced Auto Program Manager, FAA Washington, DC.....5

Luncheon: 12 Noon - 1:30pm

Speaker: MajGen Albert J. Edmonds, Deputy Director for Defense-Wide C4 Systems, Joint Staff.....6

Reuse: Domain Analysis: 2:00 - 3:30pm

Chairperson: Mr. Daniel Hocking, AIRMICS, Atlanta, GA

1.1 Validating the RAPID Center Development Methodology - E. Wright, USAISSDC, Fort Belvoir, VA.....11

1.2 Creating an Organon: Intelligent Reuse of Software Assets and Domain Knowledge - J. Solderitsch, Valley Forge Laboratories, UNISYS Defense Systems, Inc. PAoli, PA.....15

1.3 Organizing Software Around the Threads of Control- J. Cannella, P.P. Texel & Company, Eatontown, NJ.....24

Education: 2:00 - 3:30pm

Chairperson: Mr. Donald C. Fuhr, Tuskegee University, Tuskegee, AL

2.1 Teaching Ada: Lessons Learned - D. Naiditch, Hughes Radar Systems Group, Los Angeles, CA.....28

2.2 An Issue to be Considered When Revising DOD-STD-2617A-R. Abbott, California State University, Los Angeles, CA.....34

2.3 Mathematics Placement Testing: A Student Project - D. S. Martin, University of Scranton, Scranton, PA.....42

Development Methods I: 2:00 - 3:30pm

Chairperson: Ms. Susan Markel, TRW, Fairfax, VA

3.1 Maintaining Transparency of Database Objects over Networks in Ada Applications - E. Vasilescu, S. Salih, and J. Skinner, Grumman Data Systems, Woodbury, NY.....46

3.2 Synthesis of Design Methodologies for Software Development in Ada - H. J. Alters, TRW, Inc, Fairfax, VA.....55

3.3 An Ada Experiment on the Intel Hypercube - R. J. Leach, D. M. Coleman, and L. P. Tan, Howard University, Washington, DC.....68

Object Oriented: 2:00 - 3:30pm

Chairperson: Ms. Laura Veith, Rational, Bethesda, MD

4.2 Arabic OOD Methodology for Use of Ada in the Arab World - J. Agrawal and A. Al-Dhelaan, King Saud University, Riyadh, Saudi Arabia.....74

4.3 Object Coupling and Object Cohesion in Ada - E. V. Berard, Berard Software Engineering, Inc., Gaithersburg, MD

Presentation: 4:00 - 5:30pm

DISA's Roles in the Center for Information Management
Mr. Peter M. Fonash, Center for Information Management.....80

Wednesday, February 26, 1992

Exhibit Hours: 2:00 - 7:00pm

Opening Session: 8:30 - 9:00am

Keynote Speaker: Mr. Paul A. Strassmann, Director of Defense Information, OASD (C3I)

Corporate Information Management (CIM) Panel: 9:00 - 10:30am.....81
Moderator: Dr. C Fischer, OASD-C3I

Reuse: Abstract Data Types: 10:45am - 12:15pm

Chairperson: Dr. Verlynda Dobbs, Telos Corp., Shrewsbury, NJ

5.1 Engineering "Unbounded" Reusable Ada Components - J. Hollingsworth and B. W. Weide, Ohio State University, Columbus, OH.....82

5.2 Intelligent Abstract Data Types - R. A. Willis, Jr., and L. Morell, Hampton University, Hampton, VA.....98

5.3 A Reusable Ada Model for Interprocess Communication - M. J. O'Connor, Teledyne Brown Engineering, Huntsville, AL; and J. W. Hooper, Marshall University, Huntington, WV.....110

Management of Software Development: 10:45am - 12:15pm

Chairperson: Dr. George Watts, US Army CECOM, SED, Fort Monmouth, NJ

6.1 In the Trenches with Ada - P. D. Bates, USAISSDCL, Fort Lee, VA; and B. Jeffcoat, Computer Sciences Corp, Prince George, VA.....116

6.2 Small Business Efforts to be Ada Competitive - How can the Little Guy get in the Game? - J. P. Hoolihan and L. J. Corbett, J. G. Van Dyke and Associates, Columbia, MD.....125

- 6.3 Acquisition Model for the Capture and Management of Requirements for Battlefield Software Systems - H. Black, US Army CECOM, SED, Fort Monmouth, NJ.....134

Development Methods II: 10:45am - 12:15pm

Chairperson: Mr. Steve Lazernach, ALSYS, Reston, VA

- 7.2 Alternative Documentation and Review Practices - R. Metell and L. Heidelberg, US Army CECOM Software Engineering Directorate, Ft Monmouth, NJ.....139

- 7.3 Using Petri Net Reduction Techniques to Detect Ada Static Deadlocks - P. Rondogiannis and M. H. M. Cheng, University of Victoria, Victoria, B. C., Canada.....147

Student Papers: 10:45am - 12:15pm.....158

Luncheon: 12:30 - 2:00pm

Speaker: LTG Emmett Paige, Jr. US Army (Ret.), President, OAG Corporation

Reuse: Reverse Engineering: 2:15 - 3:45pm

Chairperson: Ms. Christine Braam, Centel Technology Center, Chantilly, VA

- 8.1 Don't Trash Old Code: Recycle, Renew and Convert it to Ada - J. M. Scand. a, Scandura Intelligent Systems, Narberth, PA.157

- 8.2 The Economics of Translating Space Shuttle HALS Flight Software to Ada for Reuse in Shuttle Derived Avionics Systems - J. England, Intermetrics, Inc., Huntington Beach, CA; and R. Ritchie, Rockwell International, Downey, CA.....174

- 8.3 Ada Transition Research Project (A Software Modernization Effort) - G. E. Racine and R. Hobbs, AIRMICS, Georgia Tech, Atlanta, GA; and R. Wassmuth, Fort Gilem, GA.....192

Metrics: 2:15 - 3:45pm

Chairperson: Mr. Carrington Stewart, NASA, Lyndon B. Johnson Space Center, Houston, TX

- 9.1 Design Metrics through a DIANA Based Tool - W. Zage and D. Zage, Ball State University, Muncie, IN; and D. Gaumer and M. Meier, Magnavox Government and Industrial Elec. Co., Fort Wayne, IN.....202

- 9.2 Automating Test Systems for Tactical Computer Software - J. Fowler, P. Johnson, and B. Herleth, TELOS Systems Group, Fort Sill, OK.....208

- 9.3 So Much to Measure - So Little Time to Measure it. The Need for Resource - Constrained Management Metrics Programs - S. Fenick, US Army CECOM, SED, Fort Monmouth, NJ; and H. Joiner, TELOS, Shrewsbury, NJ.....220

Software Process Improvement Panel: 2:15 - 3:45pm.....230

Moderator: Don O'Neill, Consultant

Preparing Students for Industry Panel: 4:00 - 5:30pm.....231

Moderator: Dr. Genevieve Knight, Coppin State College

Software Reuse Panel: 4:00 - 5:30pm.....232

Moderator: Dr. Harry F. Joiner, Telos Federal Systems

Reuse Business Issues Birds of a Feather: 7:30 - 8:30pm

Thursday, February 27, 1992

Reuse: Architecture: 8:30 - 10:00am

Chairperson: Dr. Charles Little, Science Application International, McLean, VA

- 10.1 An Implementation of a Generic Workstation Architecture for Command and Control Systems - K. A. Vanderstus and P. M. Richards, SofTech, Inc., Colorado Springs, CO.....233

- 10.2 A Schema for Extensible Generic Architectures - C. Meadow and L. Latour, University of Maine, Orono, ME.....247

- 10.3 Impact of System Adaptation on Generic Software Architectures - K. Gilroy, Software Compositions, Melbourne Beach, FL.....258

Ada Applications in Aeronautics & Space Systems in Development at NASA Panel: 8:30 - 10:00am.....275

Moderator: Carrington Stewart, NASA JSC

Real Time: 8:30 - 10:00am

Chairperson: Ms. Lurana Clever, Florida Institute of Technology, Melbourne, FL

- 11.1 Dynamic Configuration with Ada - R. Gerlich, Dornier, Friedrichshafen, Germany (W).....276

- 11.2 Handling Priority Inversion Problems Arising During Elaboration in Ada Programs for Real-Time Applications - L. C. Lander and S. Mitra, Thomas J. Watson School of Engineering, Applied Science and Technology, State University of New York, Binghamton, NY.....285

- 11.3 Ada Tasking Optimization Issues - A. Goel, Colts Neck, NJ; and M. Bender, US Army CECOM, SED, Fort Monmouth, NJ.....294

Ada9X Panel: 10:30am - 12 Noon.....304

Moderator: Christine Anderson, U.S. DoD, Elgin AFB, FL

Reuse: General: 10:30am - 12:00 Noon

Chairperson: Ms. Judith Giles, Intermetrics, Inc., Cambridge, MA

- 12.1 A Reusable Ada Package for Scientific Dimensional Integrity - G. W. Macpherson, Colorado Springs, CO.....305

- 12.2 Ada Software Reuse in Support of Operation Desert Storm - R. Brown, J. Morgan, and J. Labhart, Plano, TX.....319

- 12.3 Development of Cost Estimation Prototypes - J. O. Jenkins and A. J. C. Cowderoy, City University, Northampton Square, London, U.K.....323

Artificial Intelligence: 10:30am - 12:00 Noon.....331

Chairperson: Dr. Richard Kuntz, Monmouth College, W. Long Branch, NJ

- 13.1 The Development and Application of an Ada Expert System Shell - V. S. Dobbs and C. A. Burnham, TELOS Systems Group, Shrewsbury, NJ.....332

- 132 **Boiler Model: A Qualitative Model-Based Reasoning System Implemented in Ada** - J. F. Staszewski, Wintonia, TX; and Y. Lee, Naval Postgraduate School, Monterey, CA 337
- 133 **Underwater Multi-Dimensional Path Planning for the Naval Postgraduate School Autonomous Underwater Vehicle II** - Y. J. Lee, Naval Postgraduate School, Monterey, CA; and J. Bonsignore, Jr., Woodbridge, VA. 357

Luncheon: 12 Noon - 1:30pm

Speaker: Dr. Terry Straeter, Vice President & General Manager, General Dynamics-Electronics Division

Futures Panel: 2:00 - 4:30pm 351

Moderator: Miguel Carrio, MTM Engineering, Inc.

Friday, February 28, 1992

ASEET Workshops: 8:30am - 12 Noon

Ada Tasking

Ada Generics

Founders' Award



James E. Schell II
Managing Director
The HiBisCUn Corporation

James E. Schell, II is a Retired US Army Civilian and was Director of the Center for Software Engineering, CECOM at Fort Monmouth, N.J. He also served as Director, Center for Tactical Computer Systems, CECOM, and program Manager, marine Integrated Fire and Air Support System, Norden Systems, Inc., to name a few of his many responsibilities. His military experience included 2 years in the US Army Signal Corps. Mr. Schell was instrumental in the founding of the Ada Technology Conference and has given his support since its inception.

Since his retirement as a U.S. Army Civilian, Jim has served as President of SOPHSYS, Inc. and is now Managing Director of The HiBisCUn Corp.

James Schell received his BA in Math/Physics/French from Morehouse College, attended the MBA Program at California State University Northridge, CA; the Executive Development Center of Management, UC Berkeley; Brookings Institution, Executive Development; and the University of Southern California, Executive Program. He attended the US Air Force School of Electronics, Keesler AFB, MS, Radar; Ft. Monmouth Engineering Education Center, Computer Engineering; Defense Weapon Systems Management Center,

Acquisition and Management of Weapon Systems, Wright-Patterson AFB, OH; and Federal Executive Institute, Management, Arlington, VA;

Mr. Schell has received numerous honors throughout his distinguished career. These include, honorary Doctor of Laws degree from Morehouse College in 1984; Secretary of the Army Award for Most Outstanding Achievement in Equal Opportunity, 1984-85; Decorations for Meritorious Civilian Service, Dept. of the Army, 1986, 1988; Commander's Award US Army Materiel Command, for Outstanding Achievement in EEO, 1986; and Humanitarian Award, National Association of Negro Business and Professional Women's Clubs, Inc. 1986.

Mr. Schell is a member of Monmouth College Technology Advisory Council, American Defense Preparedness Assn, Armed Forces Communications-Electronics Assn., Atlanta University Science Research Institute Advisory Board, American Association for the Advancement of Science and the Planetary Society.

He is married to the former Doris Elizabeth Hunter and they have five adult children.

Keynote Speaker



Grady Booch **Director of Object-Oriented Products** **Rational**

Grady Booch has been with Rational, a company that provides advanced software engineering solutions, since its foundation in 1980.

Booch has pioneered the application of object-oriented design methods, applicable to a variety of object-based and object-oriented programming languages. His work centers primarily around very complex software systems. In particular, his design methods are being applied on systems as varied as the U.S. and European Space Station projects, the FAA Advanced Automation System, and several large command and control systems in the U.S., Europe, and Japan. He has also been actively involved in Ada research, environment implementation, and education since 1979. In 1983, Booch was given an award for distinguished service to the Ada program by the Under Secretary of Defense. Booch has served as a Distinguished Reviewer for the Ada 9X program.

Booch is the author of three books published by Benjamin/Cummings, including *Software Engineering with Ada* and *Software Components with Ada*. As a derivative work to his second book, Booch has developed class libraries in Ada and C++, each consisting of approximately 150,000 lines of source code, which are currently in production use in over 250 companies worldwide. His third book, titled *Object-Oriented Design*, describes his notation and process of object-oriented design, together with case studies in Smalltalk, Object Pascal,

C++, CLOS, and Ada. He has also published more than 50 technical articles on object-oriented design and software engineering. Booch has lectured on these topics at numerous conferences and workshops in the United States, Europe, and the Pacific Rim.

Booch is a Distinguished Graduate of the United States Air Force Academy, where he received his B. S. in Computer Science in 1977. While in the Air Force, Booch was assigned for three years at Vandenberg Air Force Base as a project director for various computer systems in support of the Shuttle and MX programs, and then assigned for two years on the faculty of the Computer Science Department of the U.S. Air Force Academy. He received an M.S.E.E. in Computer Engineering from the University of California at Santa Barbara in 1979.

Booch is a member of the American Association for Artificial Intelligence, the Association for Computing Machinery, the Institute of Electrical and Electronic Engineers, Computer Professionals for Social Responsibility, and the Classification Society of North America.

Moderator
DoD Acquisition Panel



John H. Sintic
Director
Life Cycle Software Engineering Center
U.S. Army Communications-Electronics Command
Fort Monmouth, New Jersey

John H. Sintic assumed his current position as Director, CECOM's Life Cycle Software Engineering Center on 1 April, 1988. The CECOM LCSEC is the single CECOM focal point for providing software life cycle management, software engineering and software support to 189 Mission Critical Defense Systems (MCDSs) used in strategic and tactical Battlefield Functional Areas (BFAs) supported by CECOM. The CECOM LCSEC is also the Army/Army Materiel Command focal point for Computer Resource Management, Software Technology, Ada Technology, Joint/Army Interoperability Testing and Software Quality and Productivity.

Mr. Sintic has been with the Center since December 1983. Prior to his present assignment, he served as Deputy Director of the Center. He also served as Associate Director, Computer Resource Management and Software Engineering Support, CECOM CSE. Mr. Sintic has over 27 years of experience in the field of software and computer technology.

Before joining the Center, Mr. Sintic was chief of the Engineering Division Joint Interface Test Force/Joint Interoperability

Tactical Command and Control Systems (JITF/JINTCCS) from 1978 to 1983. In this position, he directed engineers and computer scientists (military and civilian) in the Research Development and Engineering for Joint Service Communications Systems. He served as project manager for the development of the Joint Interface Test System(JITS)- the world's largest distributed command deployed to eleven Joint Service//Agency test sites.

Mr. Sintic was the recipient of the DA Achievement medal for Civilian Service in 1989 as well as the AMC Commander's Award for the 10 Outstanding AMC personnel of the Year in 1990.

Mr. Sintic has a Bachelor of Science Degree in Computer Science. He is involved in many civic functions and is currently in his seventh year on the Ocean Township Board of Education. He is also a member of the Monmouth College High Technology Advisory Board.

Mr. Sintic and his wife, Trudy, have four sons - John, Drake, Todd and Jimmy. They reside in Oakhurst New Jersey.

DoD Acquisition Panel



MG Alfred J. Mallette Commanding General US Army Communications-Electronics Command Ft. Monmouth, New Jersey

Major General Al Mallette was born in Green Bay, Wisconsin. Upon completion of the Reserve Officers Training Corps curriculum at St. Norbert in 1961, he was commissioned a second lieutenant and awarded Bachelor of Science degrees in Physics and Mathematics. He also holds a Master of Science in Operations and Research Analysis from Ohio State University. His military education includes completion of the Signal Officer Basic Course, the Infantry Advanced Course, the U.S. Army Command and General Staff College and the Industrial College of the Armed Forces.

Major General Mallette has held a variety of important command and staff positions, culminating in his current assignment as Commander, U.S. Army Communications-Electronics Command, Ft. Monmouth. Other key assignments held recently include: Commanding General, 5th Signal Command/Deputy Chief of Staff, Information Management, USAREUR; Deputy Director, Plans, Programs and Systems Directorate (DISC4), Office of the Secretary of the Army, Washington, DC; Deputy Commanding General, U.S. Army Signal School, Fort Gordon; Commander, 93rd Signal Brigade, USAREUR; and Commander 8th Signal Battalion, USAREUR.

Major General Mallette served in a variety of important career building assignments preparatory to his most recent duties. He was Chief, Program Section, Information Systems Division, Allied Forces Central Europe, Brunssum, the Netherlands; he was the S-3 and later Executive Officer, 121st Signal Battalion, 1st Infantry Division, Fort Riley; and served as Chief, Plans and Policies Development Division, U.S. Army Support Command, as well as serving in the Office of the Deputy Chief of Staff for Logistics/Criminal Investigation Division, U.S. Military Assistance Command, Vietnam.

Awards and decorations which Major General Mallette has received include the Legion of Merit (with two Oak Leaf Clusters); the Bronze Star Medal (with Oak Leaf Cluster); the Meritorious Service Medal (with Oak Leaf Cluster). Major General Mallette is also authorized to wear the Senior Parachutist Badge.

Major General Mallette and his wife, Nancy, have three children: Scott, Randy and Nicole.

DoD Acquisition Panel

Michael Perie Advanced Automation Program Manager FAA

Mike Perie is currently the Program Manager for the Advanced Automation System, Federal Aviation Administration, Washington, DC. The Advanced Automation System is a \$4.5 billion program which will be the world's largest real-time air traffic control automation system ever developed with a system life of 20-30 years. AAS will upgrade the nation's air traffic control and air navigation system. The AAS will enhance flight safety and increase flight efficiency by providing more direct and conflict-free routes which will reduce congestion and delays. The AAS represents a significant step to the future.

Mr. Perie has a Bachelor of Science Degree in Electronic Engineering from the University of Cincinnati. He is also a member of the Senior Executive Service. Mr. Perie has received a number of awards and citations for his outstanding leadership and dedication to the AAS program.

Mr. Perie was born in Hillsboro, Ohio. He is married and has two children - a son and daughter.

Luncheon Speaker



MajGen Albert J. Edmonds
Deputy Director for Defense-Wide C4 Systems
U.S. Air Force
Washington D.C.

General Edmonds received a Bachelor of Science degree in chemistry from Morris Brown College and a Master of Arts degree in counseling psychology from Hampton Institute. He entered the Air Force in 1964 and was commissioned upon graduation from Officer Training School, Lackland Air Force Base, Texas and graduated from Air War College as a distinguished graduate. He completed the National Security Program for senior officials at Harvard University in 1987.

The General was assigned to Air Force Headquarters in May 1973. As an Action Officer in the Directorate of Command, Control and Communications, he was responsible for managing Air Communications programs in the Continental United States, Alaska, Canada, South America, Greenland and Iceland. In June of 1975 the General was assigned to the Defense Communications Agency and headed the Commercial Communications Policy Office. General Edmonds was assigned to Andersen Air Force Base, Guam in 1977, as Director of Communications-Electronics for Strategic Air Command's 3rd Air Division and as commander of the 27th communications Squadron.

After completing Air War college in June 1980, he returned to Air Force headquarters as Chief of the Joint Matters Group, Directorate of Command, Control and Telecommunications, Office of the Deputy Chief of Staff Plans and Operations. From June 1, 1983 to June 14, 1983 he served as Director of Plans and Programs for the Assistant Chief of Staff for Information Systems.

General Edmonds then was assigned to Headquarters Tactical Air Command, Langley Air Force Base, as Assistant Deputy Chief of Staff for Communications and Electronics, and Vice Commander, Tactical Communications Division. In January 1985 he became Deputy Chief of Staff for Communications-Computer Systems, Tactical Air Command Headquarters, and Commander, Tactical Communications Division, Air Force Communications Command, Langley. In July 1988 he became Director of Command and Control, Communications and Computer Systems Directorate, U.S. Central Command, MacDill Air Force Base, FL. The command was responsible for U.S. military and security interests in a 19 country area in the Persian Gulf, Horn of Africa and Southwest Asia. From May 1989 until October 1990 he was Assistant Chief of Staff, Systems for Command, Control, Communications and Computers, Air Force headquarters. He assumed his present position in November 1990. His military awards and decorations include the Defense Distinguished Service Medal, Legion of Merit, Meritorious Service Medal with two oak leaf clusters, and Air Force Commendation Medal with three oak leaf clusters.

The general was named in Outstanding Young Men of America in 1973. He is a member of Kappa Delta Pi Honor Society and is a life member of the Armed Forces Communications and Electronics Association. In May 1990 he received an honorary doctor of science degree from Morris Brown College.

General Edmonds is married to the former Jacquelyn Y. McDaniel of Biloxi, Miss. They have three daughters: Gia, Sheri and Alicia.

Keynote Speaker



Paul A. Strassmann
Director of Defense Information
OASD (C3I)

Paul A. Strassmann's career includes service as chief information systems executive and vice-president of strategic planning. Since his retirement in 1985 he was an author, lecturer, consultant and university professor until his appointment in March 1991 to the position of Director of Defense Information, as a Principal Deputy Assistant Secretary of Defense (Command, Control, Communications and Intelligence). He has responsibility for managing the corporate information management (CIM) program across the Department of Defense.

He was President of Strassmann, Inc., a management consulting firm. An author and lecturer, he was also professor of management of information technology at the Imperial College, London, a member of the faculty of the Electronic University, The International School of Information Management, and professor at the Graduate School of Business, University of Connecticut.

Strassmann joined Xerox in 1969 as director of administration and information systems with worldwide responsibility for all internal Xerox computer activities. From 1972 to 1976 he served as general manager of operations, telecommunications networks, administrative services, software development and management consulting services. From 1976 to 1977 he was corporate director responsible for worldwide computer telecommunications and administrative functions. Afterwards he served as vice president of strategic planning for the Information Products Group, with responsibility for strategic investments and product plans involving the corporation's worldwide electronic

businesses. Prior to joining Xerox, Strassmann held the job of Corporate Information Officer for the General Foods Corporation and afterwards for the Kraft Corporation from 1960 through 1969. His involvement with computers dates to 1954 when he designed a method for scheduling toll collection personnel on the basis of punch card toll receipts. He earned an engineering degree from the Cooper Union, New York, and a master's degree in industrial management from MIT in Cambridge, MA. He is author of over 80 articles on information management and information worker productivity. His 1985 book Information Payoff-The Transformation of Work in the Electronic Age has attracted worldwide attention and is appearing in Japanese, Russian, Italian and Brazilian translations. His most recent book, The Business Value of Computers, shows the results of his research on the relation between information technology and profitability of firms.

Strassmann was chairman of the committee on information workers for the White House conference on productivity and served on the Department of Defense Federal Advisory Board for Information Management. He is a life member of the Data Processing Management Association, fellow of the British Computer Society, and senior member of the IEEE. He authored the code of conduct and of professional practices for the certificate in data processing.

Strassmann was a member of a guerilla commando group of the Czechoslovak Army engaged in eight months of combat ending in March 1945.

Luncheon Speaker



**LTG Emmett Paige, Jr.
U.S. Army (RET)**

Lieutenant General Emmett Paige Jr. retired from the U.S. Army on August 1, 1988 after almost 41 years of active service. He enlisted in the Army in August 1947 as a Private at the age of 16, dropping out of high school to do so.

Throughout his military career, he was in the Communications-Electronics business after completion of basic training. He completed Signal Corps Officers Candidate School in July 1952 receiving his commission as a 2nd Lieutenant.

One of his most notable assignments was Project Manager for the Integrated Wideband Communications System, Southeast Asia during the Vietnam conflict. It was the largest communications system ever installed in a combat theater covering all of South Vietnam and Thailand. He later commanded the 361st Signal Battalion in Vietnam. He served two tours with the Defense Communications Agency.

As a Colonel he commanded the 11th Signal Group at Fort Huachuca, Arizona. In 1976 he was selected for promotion to Brigadier General and Command of the U.S. Army Communications-Electronics Engineering and Installation Agency at Fort Huachuca, AZ and concurrently the U.S. Army Communications Systems Agency at Fort Monmouth, New Jersey.

In 1979 he was promoted to Major General and assumed command of the U.S. Army Communications Research and development Command at Fort Monmouth, New Jersey.

In 1981 he assumed command of the U.S. Army Electronics R and D Command, at the Harry Diamond laboratory, Adelphi, Maryland. In 1984 he was promoted to Lieutenant General and assumed command of the U.S. Army Information Systems Command where he served until his retirement in 1988.

General Paige is a graduate of the U.S. Army War College. He has been honored as a "Distinguished Alumnus" of both the University of Maryland, University College, where he obtained his Bachelors degree, and Penn State where he received his Masters degree. He was awarded an honorary Doctor of Law from Tougaloo College, Tougaloo, MS.

He is the holder of the Army Commendation Medal, the Meritorious Service Medal, the Bronze Star for Meritorious Service in Vietnam, the legion of Merit with two oak leaf clusters (3 awards), and the Distinguished Service Medal with one oak leaf cluster (2 awards). All of these are awards for outstanding service. He was selected as the Chief Information Officer of the Year in 1987 by Information Week Magazine. He was selected as the coveted Distinguished Information Sciences Award (DISA) winner by DPMA in 1988 for outstanding service to advancements in the field of Information Sciences.

LTG Paige is now the President and COO of OAO Corporation, an Aerospace and Information Services company, with Headquarters at Greenbelt, Maryland.

Keynote Speaker



Dr. Terry A. Straeter
Corporate Vice President and General Manager
General Dynamics Electronics Division

Dr. Straeter has been with General Dynamics since March of 1979. He began as Director of Technical Software and was responsible for defining the General Dynamics software technology program. He then became Vice President and Director at General Dynamics Data Systems Division, Western Center where he was responsible for all data processing and computer support for 4 General Dynamics Divisions. He then moved to the Electronics Division where he was Vice President and Programs Director, Tactical Systems from June 1983 until January 1987, Division Vice President and Assistant General Manager from January 1987 until February 1991. In February 1991 he assumed the position he now holds. This division designs, develops and manufactures military electronic systems to support tactical and strategic weapons fielded by domestic and international customers. The major products are comprised of avionics automated test systems, digital imagery workstations and secure communications equipment. These systems supported the highly successful Desert Storm activities.

Before joining General Dynamics, Dr. Straeter held a series of positions at NASA Langley Research Center where his responsibilities included optimization, air traffic control, digital flight controls and avionics, embedded software development, software technology, software and general management.

Dr. Straeter received his PhD in Applied Math from North Carolina State University in 1971, his M.A. in Math from William & Mary and his A.B. in Math from William Jewell College.

He is a member of AIAA and served on the technical committee on computer Systems from 1977 until 1980. He is also a member of IEEE, where he was General Chair of the International Conference on Software Engineering in 1984. He has produced more than 30 publications in technical journals, and has refereed conferences.

DoD AQUISITION PANEL

**Moderator: Mr. John H. Sintic, Director, Software Engineering, US Army
CECOM**

**Panelists: MG Alfred J. Mallette, Commanding General, United States
Army Communications-Electronics Command
Mr. Michael Perie, Advanced Auto Program Manager, FAA**

VALIDATING THE ARMY REUSE CENTER DEVELOPMENT METHODOLOGY

Elena Wright

U.S. Army Information Systems Software Development Center - Washington
Attention: ASQB-IWS-R (STOP H-4)
Fort Belvoir, Virginia 22060-5456
DSN 356-9071 Commercial (703) 285-9071

ABSTRACT

This paper presents a brief description of the mission, functions, and services of the Army Reuse Center, and a specific example of how one service product of the center was applied to a specific system. This system was recreated using an underlying discipline of the Army Reuse Center domain analysis, the object oriented analysis and design methodologies. These methodologies were validated in a COBOL system redesigned and implemented in Ada. The system chosen for this project was the Army's Retired Army personnel System (RAPS).

INTRODUCTION

The mission of the Army Reuse Center conforms to the Department of Defense (DoD) objectives for the use of the Ada programming language and the development of adaptable, reusable, reliable, maintainable, high quality, and cost-effective systems. The center facilitates these objectives by administering and providing an operational comprehensive reuse program focused on the needs of Army Management Information Systems (MIS) but whose tenets can be applied to all domains.

Center services include implementing policies and procedures to acquire, prepare, and certify reusable components to Army Reuse Center standards in order to install these

components into the Reuse Center Library (RCL); operating the RCL tool with facilities for classifying, storing, retrieving, and maintaining reusable components; populating the RCL with high-quality components needed for software systems (design, implementation, etc.); training all user levels from executive to programmer in the efficient use of the RCL; preparing and gathering reuse metrics; providing telephonic customer information services; and training, performing, and advising on domain analysis and utilizing reusable products. The Army Reuse Center at Software Development Center - Washington (SDC-W) is the designated Defense Information System Agency (DISA) Corporate Information Management (CIM) site for reuse support for all Army software developments.

DOMAIN ANALYSIS, AS WELL AS OBJECT-ORIENTED SYSTEM ANALYSIS AND DESIGN

Domain analysis, as well as object oriented system analysis and design methods, expedites the identification of reuse opportunities early in the software development life cycle. The Army Reuse Center object oriented systems analysis methodology combines the best of multiple methods and essentially consists of three phases; gathering system's information, systems analysis, product/model/analysis and knowledge/taxonomy generation.

METHODOLOGY PHASES

Based on the information gathered, the domain is defined with the domain boundaries, and the interfaces and scope (halting conditions), identified. During the analysis phase, domain-specific information is gathered by examining written material and interviewing domain experts. Object decomposition identifies objects the system needs in order to perform its responsibilities. Object characteristics and functionalities are defined within each object. Subsequently, the objects are identified as either classes or subclasses. The classes define the common features of the objects; the subclasses define features specific to that object. The relationship between objects may be represented as both generalization-specialization and whole-part structures. The interactions between objects are represented with messages between the objects. The domain model and taxonomy summarizes the information acquired from the above analysis. This stage culminates in codifying the knowledge gained in the analysis phase, e.g., identified and documented data sources, data sinks, and data transformation processes. The product generation phases produces the domain model, reusable product encapsulations (i.e., recurring domain abstractions), recommended classification scheme terms, and a list of recommended candidate reusable components which are all part of the Object Oriented Analysis product.

CONCEPTS VALIDATION WITH THE RETIRED ARMY PERSONNEL SYSTEM (RAPS)

The Army Reuse Center tested the validity of its domain analysis, object oriented analysis, and design methods on the Army's Retired Army Personnel System (RAPS). RAPS is a

batch and potentially interactive report generator system that provides the name, grade, address, and other basic personnel data on all retired Army personnel residing within an installation's geographical area of responsibility. Other personnel information includes data on the widows/widowers and dependents of deceased retired personnel, and widows/widowers and dependents of individuals who died while on active duty who were eligible for retirement. The reports generated by RAPS are in the form of mailing labels or rosters. The report and personnel selection criteria are based on control cards read in batch mode.

TASK REQUIREMENTS

Requirements for the RAPS redesign included building for reuse at the system and component levels, identifying and using available reuse components, taking advantage of the reuse and object oriented capabilities offered by the Ada programming language, and demonstrating portability of a system developed on a VAX/VMS environment targeting to a UNIX/Sperry environment. Personnel assigned to support this task included one RAPS systems analyst and one Army Reuse Center software engineer.

TASK IMPLEMENTATION

The new Ada system was developed with a combination of four newly developed independent, self-contained components plus the system's driver adhering to Army Reuse Center standards and four Reuse Center Reusable Software Components (RSC) extracted from the Reuse Center Library (RCL) and integrated into the system design implementation. A portable test suite for each of the four components was also developed to

validate the system under different environments. The resulting system is a successfully ported, highly modular, 10,000 lines of Ada program (including comments and blank lines) that can be reused as a complete system or as individual components. Other products of this effort are: (1) An object-oriented analysis document which included: an overview of the domain analysis, information on how requirements analysis was performed, descriptions of the analysis model, objects, and reuse potential within the system, and a system specification; (2) A generic object-oriented analysis model; (3) A design model based on Buhr notation; (4) Reuse metrics to reflect time savings; and (5) a mapping between analysis and system requirements.

VALIDATED CONCEPTS

The metrics collected throughout the life-cycle proved both the validity of the methodology and the time saved from reusing certified components. The two analysts working on the project were instructed to track and assign each work-hour to the appropriate life-cycle phase. The largest savings were found in the object-oriented design phase and the Ada implementation and unit testing phase because of the reuse of design and implementation components and test suites.

Since RAPS is a small sub-domain of a Management Information System (MIS) personnel system domain, the scope of the domain analysis was restricted from the start. The domain analysis required nine days and the object oriented analysis phase required 33 days. The object oriented design phase consumed 36.25 days in comparison with the 70 days that would have been required without access to good reusable components. The Ada implementation

and unit testing phases required only 22 days instead of the 57.06 days that would have been needed without reuse. The integration and system testing phase required 15 days. From a percentile viewpoint, 48.7% of total design time was saved; 63.9% of total coding time was saved; and 59.8% of total unit testing time was saved. Overall, considering all the development phases (analysis, design, coding, unit testing and system testing), the total time saved was 37.4%. This saving is a direct result of not having to design and code the four components extracted from the library. System integration was simplified because the extracted components were developed with reuse in mind and had minimal and identified environmental dependencies.

CONCLUSIONS

The time saved from reuse and using the Army Reuse Center domain analysis, object oriented analysis, and design methodologies is only the beginning. The Ada RAPS system, as a whole, as well as the four newly developed components, have completed the Army Reuse Center certification process and are installed in the library. This system and its components are available for all Army Reuse Center customers now. As each part is reused in new generations of software, in multiple systems, savings will multiply. Through reuse, software system development in DoD is following the lead of American manufacturing. We have left the age of planned obsolescence and can now produce the building blocks needed for well-built, high quality, reusable systems amenable to the open systems environment and serving the nation well into the next century.

REFERENCES

Vitaletti, William and Ernesto Guerrieri, Ph.D. Domain Analysis Within the ISEC RAPID Center. Proceedings of the Eighth Annual National Conference on Ada Technology: 1990.

ABOUT THE AUTHOR

Elena Wright is Chief, Domain Engineering Branch, Army Reuse Center. She holds a BA from the University of the State of New York and a MS from Strayer College. Ms. Wright is a 1985 graduate of the U.S. Army's computer programmer intern program. She has a special interest in the practical application of computer science theory to real-world problems. Other interests include Western philosophy, religion, and amateur art.

Creating an Organon – Intelligent Reuse of Software Assets and Domain Knowledge*

James Solderitsch
Paramax Systems Corporation
Paoli, PA 19301-0517[†]

Abstract

This paper briefly sketches the state-of-practice in the production of large complex systems. Methods and techniques to produce large systems can be viewed on a continuum ranging from pure manual construction with no tool based support to a highly evolved environment that greatly aids in the engineering of these systems. An organon represents one view of such an environment. The paper describes work underway at Paramax Systems Corporation, a Unisys Company, partially supported by the DARPA/SISTO STARS Program, to advance the state-of-practice on the path to such an organon.

1 Introduction

Paramax and its affiliates participating in the STARS¹ (Software Technology for Adaptable, Reliable Systems) program are keenly interested in understanding, developing (or acquiring) and applying technology to support the development of complex software systems based on a reuse perspective. The word reuse is often interpreted too strictly to mean the reuse of code components, whether informally through ad hoc reuse by a programmer remembering and reusing a previously composed code fragment, or more formally by finding and retrieving code from a code library.

In the remainder of this paper, reuse should be understood in a larger context to mean the reuse of knowledge about the complex application areas (or domains) to be served by software systems for these areas. Engineering models that capture knowledge about these

application domains can provide a vital basis for technology to support reuse. Such domain models themselves require their own enabling technology so that they can effectively infuse the application development process. Paramax believes that a knowledge-based approach to the creation of domain models can provide a successful strategy for moving the engineering of software systems further along an important maturity scale or continuum.

Techniques for the engineering of software systems can be viewed on a continuum ranging from the production of a system via ad hoc custom design and handwritten code to the automatic generation of a system from a high-level specification in a domain-specific language. In the past, knowledge gained from system development and deployment has been left in the heads of expert software developers, and has not been extracted and collected for corporate reuse. Thus there has been little movement on the maturity continuum. The development and application of domain models can accelerate this movement and thereby boost system quality and system development productivity.

2 A Domain-Specific Approach

At Paramax, we believe that the key to dramatic improvements in productivity rests in effective reuse of application domain knowledge and not just the reuse of code and other artifacts of system development. In this context, a *domain* is comprised of a set of existing and anticipated software applications that provide a common function or similar capability. Domains can be further sub-divided into horizontal and vertical domains. A horizontal domain is one (e.g. common data structure definitions and operations) whose contents intersect with vertical domains oriented around a company's line-of-business (LOB) or specialized applications area.

*Parts of this paper are adapted from a position paper accepted for the Workshop on Domain Modeling for Software Engineering held at ICSE-13, Austin TX, 13-May-1991

[†]E-mail: jjs@prc.unisys.com, or by phone at 215-648-2831

¹Paramax is supported in STARS under contract number: F19628-88-D-0031.

History has shown that many of the past success stories for reuse have come within certain well-defined domains (e.g. mathematics routines). Paramax believes that the impact and successful application of a reuse-based approach to software design and production will be greatest for (vertical) domain-specific libraries. For example, a greater proportion of a typical application can be built from parts withdrawn from such a library. There is also a higher expectation that systems built from such parts will have a closer functional fit and be more efficient. The capability exists for reusable sub-systems to be created via part selection and configuration.

There are real costs in establishing such a library and not every domain is mature and stable enough to support such an intensive reuse-based approach. Domain analysis [PD87] to support such libraries can be hard, and is certainly expensive and time-consuming. However, domain analysis is a fundamental prerequisite for a reuse environment to support the extended life-cycle of an application domain. Such support is analogous to the way that some software engineering environments support the traditional waterfall life-cycle. The goal of domain analysis is to provide fundamental support for the organized growth and development of software applications for the domain, both from the consuming side and producing side of the software equation.

To achieve this level of support, system/software development techniques and support environments must be re-oriented to directly support the creation, evolution and usage of domain knowledge. System developers must work from a common model of the application domain so that specific project requirements can be tailored based on generic knowledge gained from similar programs in the domain. We recognize that distinct activities within the system development process need to establish application domain models and reusable components. Examples of such activities include:

- requirements elicitation, refinement and verification
- automated program generation
- reverse engineering

The modeling process can serve to identify not just potential reusable components but also opportunities for application-specific common interfaces or protocols, and application parts generators. In fact, the interplay between domain analysis/modeling and system development defines a synergistic relationship that extends across an extended domain life-cycle. While

the structure of this life-cycle is still being debated, its characteristics are more cyclical (or spiral) rather than linear (such as the typical waterfall life-cycle model).

Domains that possess a high degree of cohesiveness and controlled variability can be served by a software generation approach whereby new application software can be generated from high-level specifications written in specially constructed domain-specific languages. Alternatively, these specifications can be obtained from the manipulation of a graphical representation of domain-specific artifacts. Such systems as User Interface Management Systems (UIMS) are a case in point.

Domains which are less amenable to software generation can be supported through a "knowledge base". A knowledge base captures important facets and processes that characterize software systems previously developed for the domain. Such knowledge bases can be created and evolved while such systems are maintained or reverse-engineered as a system undergoes more radical changes. If the domain model is sufficiently mature so that it takes the form of a generic system architecture, it may be possible to support the (semi-)automated construction of systems via a guided walk-through of the modeled architecture. Selecting parameter values during the walk-through can specialize system components or templates which are stored in relation to the architecture. In a less mature domain, the human designer would need to operate more autonomously, relying on the availability of components and their hand-tailorability.

A crucial component of a model based approach to software engineering is providing a *machine readable and processable representation* of the model that is capable of keeping up with the dynamic nature of the software systems whose construction the models must ultimately support. To have maximal effect, the model and a family of tools that are empowered to process and interpret the model should be blended together into a domain specific environment that serves to integrate and amplify the support and services provided by the model and tools. Moreover, the model, the representation of the model and any tools which operate on the model must all be tailored to support the operational goals implicit in the environment.

3 The Organon Concept

Paramax, through its Reusability Library Framework² (RLF [MC89]) project and related efforts, is working toward the development of an *organon*. An organon[Sim88] is the culmination of RLF and related technologies evolved and applied to support the efficient, cost-effective production of software systems. From the dictionary, an organon is defined to be

... an instrument for acquiring knowledge; specifically, a body of methodological doctrine comprising principles for scientific and philosophical procedure and investigation

An organon will

- be an interactive and evolving public storehouse of expertise and componentry serving particular application domains;
- effectively support wide-spectrum reuse including requirements, design and test cases;
- support a number of different points-of-view on system engineering including those of construction and generation; and,
- be a central repository of domain expertise that effectively combines people, plus emerging and maturing methods, plus supporting technology.

A fully-realized organon lies at the end of the continuum described earlier. Paramax is currently working to lay a foundation to move the state-of-practice further along the model-based maturity scale. Paramax is acquiring and producing a technology base to provide a machine readable and processable representation of domain models in a form which domain-specific tools may directly utilize and manipulate. The RLF is a set of Ada knowledge-based tools (semantic network and rule-based systems that can be used in concert) to support the definition and manipulation of domain models. The RLF has been used to develop several reuse libraries [SWT89] as well as a model-based tool utilization assistant (TUA) for the domain of document preparation. An early version of the RLF was used to produce an Ada Unit Test Assistant (Gadfly) [WSS+88] which contained a model of test heuristics and generated test plans based on parsing of Ada units and interaction with a human test engineer.

²The development of the RLF began as a STARS Foundations project, contract number N00014-88-C-2052, administered by the Naval Research Laboratory and is now supported under the aforementioned STARS contract.

3.1 The Constructive Approach

In its current form, the RLF promotes the use of a visible domain model in the creation and operation of reuse libraries for particular application domains. As such, the RLF explicitly addresses the construction point-of-view. The domain model, captured by the RLF in terms of its semantic network and inferencing subsystems, becomes a palpable part of the interface to the library system and can educate novice library users and accelerate the productivity of expert users. The library, through its dynamic representation of knowledge about the domain, becomes much more than a static collection of components. RLF features and capabilities are being enhanced over time to support library content evolution (e.g., replace family of part variants with a suitable generator); automatic maintenance of library content and persistent user models; and, automatic solicitation for new components to cover gaps in library coverage.

Moreover, the RLF is able to provide support for the computer-aided construction of software systems from parts present in the library. Such parts may be Ada source files, source template files, test cases, design diagrams or even requirements. Inferencing techniques can be applied to help decide which parts are needed for a system under construction and to determine how the parts should be tailored or adapted. If the kind of adaptation is parameterizable, final part configurations can be generated from templates or specifications.

The RLF supports a constructive approach to domain specific software architectures (so-called DSSAs). In the paper [D'I89], D'Ippolito expands on the notion of models for the description of different domain specific software architectures. He discusses methods for storing and retrieving domain models and uses the familiar description of a system in terms of its parts as well as the specialization of certain parts as kinds of other parts. The AdaKNET semantic network subsystem of the RLF was expressly designed to represent this sort of information in a form that lends itself to graphical presentation and interactive examination.

3.2 The Generative Approach

The generative approach is reflected in the Paramax IR&D work within the Program Generation Techniques (PGT) and Application Specific Languages (ASL) projects. PGT produced Ada-based meta-generation technology [PKP+82] that enables the cost-

effective and efficient production of software systems from domain specific specifications of their essential properties.

ASLs are particular realizations of the power of the PGT generation system. Perhaps the best representative of the potential of the ASL approach is seen in the Message Format Processing Language (MFPL) [PSL87]. The construction of the MFPL processor, and the creation of a specification of a single military message format in MFPL, was no more expensive than custom-building the message processing software for the single format by hand. However, once the MFPL translator was built, new or modified message formats could be processed by software generated through MFPL. Specifications can be created or modified in days (or even hours) contrary to manual construction or modification of source code which often took months.

The MFPL work is related to work performed at the SEI [PL89]. In particular, the paper by Plinta and Lee addresses message formatting and uses an approach based on the notion of typecasters which provide a model for the message formatting domain. Whereas this approach ultimately requires a human programmer to tailor and produce code based on templates, the use of generation provides an effective means by which productivity, accuracy and ultimately quality can be increased beyond that achieved directly from the typecaster model. A variant of MFPL could provide the enabling technology to enhance this model based approach.

4 The RLF System

The essential feature of an organon is its role in capturing relevant knowledge about families of related systems, and making this information available to help the development of new or enhanced systems. The RLF project, despite its origins in the support of software library systems (and despite its name), is vitally concerned with this feature.

Many current reuse library systems are built on the faceted classification scheme [PD91]. There are a number of limitations with the implementations of this approach. Among them are

- reliance on a query specification and refinement approach to discover the contents of the underlying software catalog
- lack of changeability of the underlying classification scheme as the domain evolves

- no explicit support for supporting different user communities (e.g. managers and programmers) and different user abilities
- lack of a graphical view of the underlying domain model.

The knowledge-based approach taken in the RLF provides a set of capabilities that include virtually all of those provided by a faceted classification system. The RLF permits a library organization to evolve along with the components being kept in the library and is better able to support software collections as they change both in size and maturity. An adaptable library organization is better able to serve the needs of focused application domains. Important semantic attributes of software assets are often dependent on the domains to which they belong. As such, library support software must be semantically tailorable to represent and use such attributes.

The RLF seeks to overcome some of the weaknesses apparent in other classification-based reuse support systems. One important aspect is the accessibility of the classification scheme itself and the relative ease by which the classification data base can be tailored and extended. Moreover, the RLF provides the user with guidance on the use of the classification system so that the user is not forced to become an expert in the classification scheme to use it effectively.

Inspiration for the RLF was provided by the KL-ONE system of Ron Brachman [BS85] and a Prolog descendent developed at Paramax [Mat87]. The RLF is also related to a number of other on-going research projects including LaSSIE [DBSB91] and AIRS [OH87]. The RLF is based on three key components. The first is a semantic network formalism which supports both multiple inheritance (classes having multiple parents) and multiple individuation (individuals belonging to multiple classes). The second is a rule-based inferencing capability which is integrated with the semantic network. Inference bases are localized at concepts which lie in the network and an inferencing session can span multiple inference bases within the network. The third component is association of additional information as state attached to concepts within the network. Such state information is not inherited but has proven useful in providing richer domain representations. Possible state values include text files and graphical design data.

Broad objectives of the RLF project include:

- develop knowledge-based interfaces to asset library (object) management systems;

- investigate the mapping between application domain and reuse technology (part selection, part composition, part generation, including computer-assisted versions of each of these approaches);
- go beyond supporting retrieval of static parts to include program generation, system/software configuration, system/software testing and even system/software design and requirements analysis;
- support the basic integration of reuse technologies (knowledge-based and generation techniques); and,
- perform some applied research in domain analysis.

4.1 Major RLF Subsystems

Figure 1 illustrates the basic composition of the RLF's major subsystems. All components of this system were developed from an Ada perspective using basic principles of data abstraction, information hiding and strong typing.

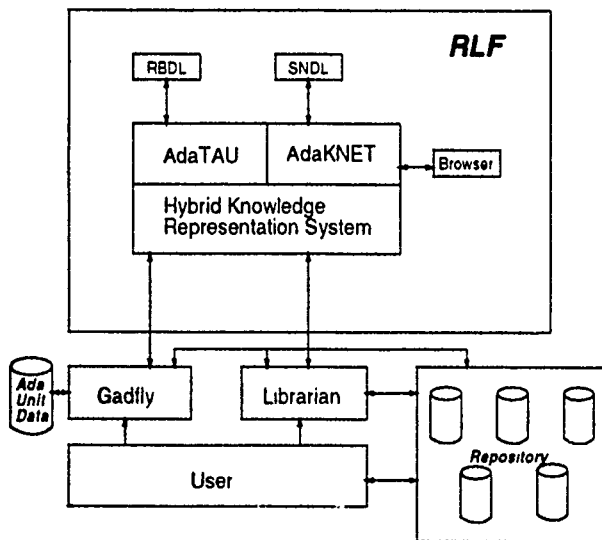


Figure 1: RLF Overview

Abstract data types were produced after analyzing the structure of proven Knowledge Representation Systems (KRSs), first by focusing on the operations provided by these systems, and only later considering possible internal representations of knowledge held within the system. No attempt was made to naively import features native to AI programming language paradigms such as pattern matching or theorem proving.

AdaKNET is a semantic network system useful for capturing static information describing the basic state of some enterprise or subject area. For example, our two initial uses of AdaKNET were to capture some basic Ada semantics regarding Ada compilation unit structure and portions of the Ada type lattice for use by Gadfly, an Ada unit test plan generator; and, to represent some basic relationships among Ada benchmark programs for use in an Ada benchmark program library system. An important part of our work concerns how to combine the representational power of AdaKNET with other systems, including other KRSs.

AdaTAU is a rule base system that can be used as a stand-alone system or in conjunction with other knowledge representation systems such as AdaKNET. Rules collected into rule bases are used to infer new facts from a collection of initial facts. New knowledge is added to a system employing the facilities of AdaTAU so that AdaTAU is acting like an expert system that enhances the capabilities of the original system. When used together with a system like AdaKNET, AdaTAU becomes part of a hybrid KRS where the role of AdaTAU is to facilitate the capture and use of dynamic information that is normally outside the realm of the other cooperating KRS. For example, the benchmark librarian rules are used to advise librarian users of operational information regarding benchmark components that are not easily discernible within the benchmark taxonomy provided through AdaKNET.

A careful separation of the content of knowledge bases from their basic organization and available operations is provided through the use of two specification languages developed explicitly for the RLF (cf. [SWT89]). RBDL (Rule Base Description Language) and SDDL (Semantic Network Description Language) are used to specify rule and fact base descriptions for AdaTAU and semantic network descriptions for AdaKNET respectively. Individual knowledge base definitions are translated automatically to an Ada compilation unit that, when executed, produces a machine readable version of the original specifications. The design and implementation of these specification languages was accomplished through the use of the meta-generation system cited earlier in conjunction with MFPL.

The end user typically works directly with an application built on top of AdaKNET, AdaTAU or a hybrid of both of them. In addition, an application makes use of its own data structures. For example, in using the Gadfly application, knowledge about an Ada unit under test is assembled and stored within a hybrid knowledge base. From this knowledge gained by

examining the Ada unit directly and as a result of a dialogue conducted with the user, suggested test case plans are generated for the user. For the librarian user, a collection of Ada modules is available for direct examination. Alternatively, the user can browse, or be "expertly" guided through, an information web that captures essential information about the contents of the library. A library user offering a new component for the library can be guided to the right insertion point and, using an integrated form of Gadfly, be advised of necessary quality control measures to be taken before the component can be officially installed.

4.2 RLF Graphical Browser

Recently, an X11R3-based graphical browser interface was developed for librarian applications built on top of the RLF. Figure 2 is a screen dump of a browser session in progress. There are several things to note about this image.

Boxes indicate AdaKNET concepts. Thin lines show the specialization hierarchy provided in the model. Thus all of the boxes derived from process are kinds of processes. The model shown in the larger graphical display is a partial view of an AdaKNET representation of a domain-specific, reused-based process model which was prototyped recently. In addition to the previously mentioned applications of the RLF, there has been some work on using the hybrid knowledge representation capabilities to provide process model representation and enactment.

Only the specialization (IS-A) structure is shown graphically. The small subwindow in the lower right is a list of the AdaKNET roles (or attributes) of the model.domain concept. This window is drawn in response to a menu choice that is offered to the user when she presses the mouse button on a concept in the full-size window. For example, every individual type model.domain can have a number of agents, exactly one parent process, a number of preconditions, etc. The elongated window on the right shows a reduced view of the model displayed in the larger window. The scroll bars are used to pan around within either review. Any motion within one window causes the corresponding motion to be made in the other window.

There are also a number of choices provided to the user through a menu bar at the top of the window. Additional work on graphical browsers for RLF models is currently underway. Recently, a version of the RLF graphical browser has been produced which allows AdaTAU inferencer interactions to take place in

another window with the graphical browser automatically scrolling to other portions of the model as a result of user interactions with the inferencer.

5 Advanced Knowledge-Based Library System

Paramax believes that the RLF, or technology derived or analogous to it, can form the basis for a next generation library system. Figure 3 shows a view of such a system. A library system with the properties indicated in the figure represents another step along the software engineering continuum.

As shown in the figure, there are multiple means to access the library system which is shown as an interconnected series of distributed domain-specific libraries. The user can pose queries, manually browse the system through a graphical browser, or be guided by an intelligent library assistant. Each physically distributed library may be equipped with its own library domain model. In addition, an umbrella library model will be accessible to the user at her workstation to present a seamless view into the distributed system. Additional library support services provide detailed user models that capture and retain information about user classes and user usage histories.

Current effort within the RLF project is directed to accomplishing this advanced library system. For example, activities are underway to employ the Andrew File System (AFS) [SK89] to provide truly distributed libraries. Experiments are being conducted in the areas of library asset interchange so that assets may be freely distributed among cooperating libraries. The RLF already supports all three of the interaction paradigms shown in the figure.

6 Conclusion

This extended abstract sketches an approach to moving the state-of-practice of producing software systems further along a maturity scale which has been dubbed the software engineering continuum. The end of this spectrum has been termed an organon wherein there is a full environment of tools to support the system engineer in her task of producing complex software systems. There is still a wide gap between the current

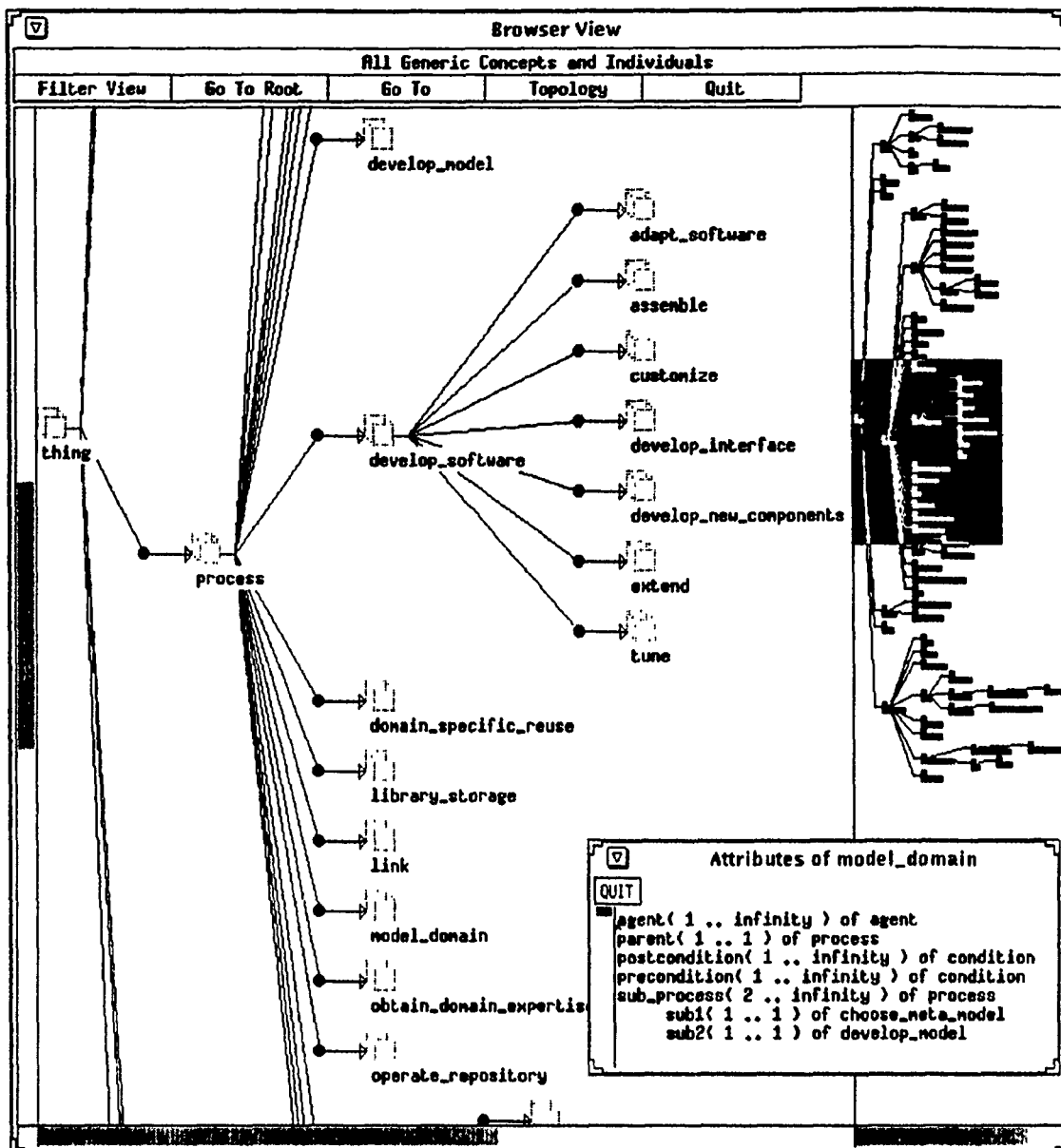


Figure 2: RLF Browser

state of practice and the organon goal. The RLF³ project and others like it (e.g. LaSSIE) are providing opportunities for reduce this gap. By actually building and maintaining application systems using such environments, we will be able to identify shortcomings in individual approaches and merge related ones. The key to success is provided by taking knowledge-based techniques and applying them in novel and useful ways to the construction and generation of high quality and

economical systems.

7 References

- [BS85] Ronald J. Brachman and J. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171-216, 1985.
- [DBSB91] Premkumar Devanbu, Ronald J. Brachman, Peter G. Selfridge, and Bruce W. Ballard. Lassie: A knowledge-based software

³RLF version 2.2 is currently approved for general public release and is available via anonymous FTP at the internet address stars.rosslyn.unisys.com, or by tape from Paramax/STARSCenter, Reston, VA

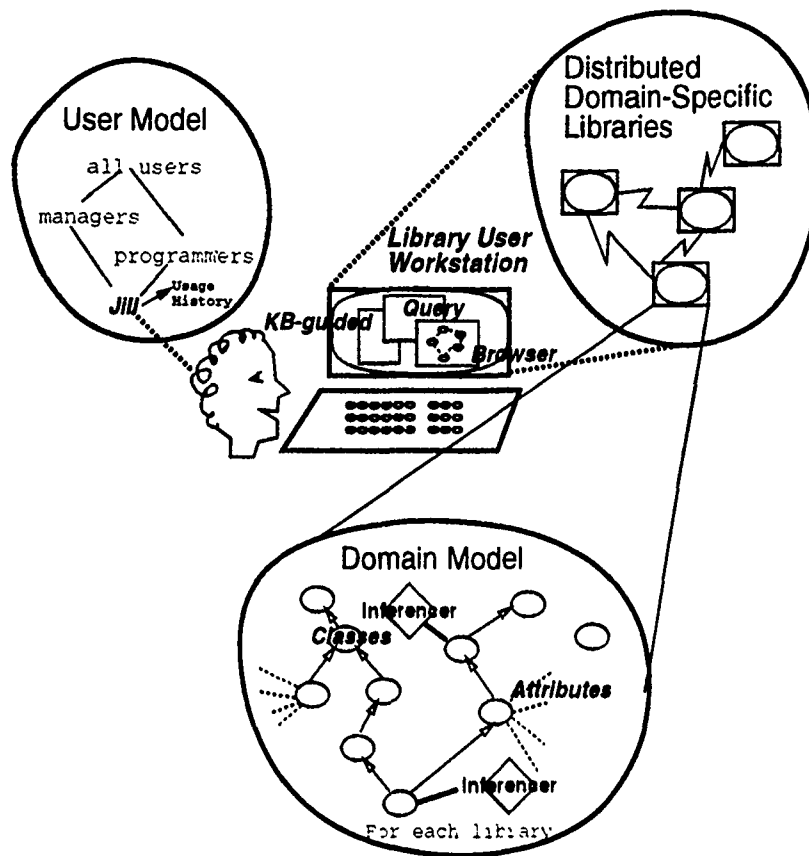


Figure 3: Advanced Library System

- information system. *Communications of the ACM*, 34(5):34-49, 1991.
- [D'89] R. D'Ippolito. Using Models in Software Engineering. In *Proceedings: TRI-Ada '89*. ACM, October 1989.
- [Mat87] D. L. Matuszek. K-Pack: A Programmer's Interface to KNET, October 1987. Logic-Based Systems Technical Memo 61.
- [MC89] R. McDowell and K. Cassell. The RLF Librarian: A Reusability Librarian Based on Cooperating Knowledge-Based Systems. In *Proceedings of RADC 4th Annual Knowledge-Based Software Assistant Conference*, September 1989.
- [OH87] E. Ostertag and J. A. Hendler. AIRS: An AI-based Ada reuse system, 1987. Tech. Rep. CS-TR 2197, University of Maryland.
- [PD87] R. Prieto-Diaz. Domain Analysis for Reusability. In *Proceedings of COMPSAC 87*, October 1987.
- [PD91] Ruben Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88-97, 1991.
- [PKP+82] Teri F. Payton, S. E. Keller, John A. Perkins, S. Rowan, and Susan P. Mardinly. SSAGS: A Syntax and Semantics Analysis and Generation System. In *Proceedings of COMPSAC '82*. IEEE, 1982.
- [PL89] C. Plinta and K. Lee. A Model Solution for the C⁴I Domain. In *Proceedings: TRI-Ada '89*. ACM, October 1989.
- [PSL87] R. Pollack, J. Solderitsch, and W. Loftus. A Generative Approach to Message Format Processing. In *Proceedings: Unisys Software Engineering Symposium*, September 1987.
- [Sim88] M. Simos. The Growing of an Organon: A Hybrid Knowledge-based Technology and Methodology for Software Reuse. In *Proceedings of the National Conference on*

Software Reusability, April 1988.

- [SK89] A. Z. Spector and M. L. Kazar. Uniting File Systems. *Unix Review*, 7(3), March 1989.
- [SWT89] J. Solderitsch, K. Wallnau, and J. Thahamer. Constructing Domain-Specific Ada Reuse Libraries. In *Proceedings, 7th Annual National Conference on Ada Technology*, March 1989. pages 419-433.
- [WSS⁺88] K. Wallnau, J. Solderitsch, M. Simos, R. McDowell, K. Cassell, and D. Campbell. Construction of Knowledge-Based Components and Applications in Ada. In *Proceedings of AIDA-88, Fourth Annual Conference on Artificial Intelligence & Ada*. George Mason Univ., November 1988.

Biography

James Solderitsch was first employed by Paramax in 1986 after having been an Assistant Professor in the Mathematical Sciences Department of Villanova University. Since coming to Paramax, he has worked on the design and development of Application-Specific Languages (ASLs) using compiler-compiler technology and currently is serving as technical lead for the STARS-sponsored task concentrating on reuse within Valley Forge Laboratories. His active research interests include software reuse, domain analysis, knowledge representation issues arising from domain analysis and developing and maintaining domain specific software architectures. He received his Ph. D. in Mathematics from Lehigh University in 1977. He may be reached at the address Paramax Systems Corporation, 70 E. Swedesford Road, Paoli, PA 19301-0517. His telephone number is 215-648-2831 and his electronic mail address is jjjs@prc.unisys.com.

ORGANIZING SOFTWARE AROUND THREADS OF CONTROL

John K. Cannella
P.P. Texel & Company, Inc.
Eatontown, N.J. 07724

Abstract

During the past few years, ever since DOD-STD-2167A was adopted, there have been many debates over the proper definitions of Computer Software Components (CSCs) and Computer Software Units (CSUs). 2167A was written with a bias toward a process-oriented definition of a CSC. This paper provides a CSC definition that is process-oriented and justifies the definition in terms of 2167A. It will also show how this process-oriented definition does not preclude using object-oriented methodologies.

Introduction

Software Preliminary Design is the identification of the modules in a software system and the identification of the interfaces between these modules. One widely accepted design methodology is Object Oriented Design (OOD).

OOD with Ada provides a method of decomposing a software system into Ada packages. Each Ada package represents an object. These objects have relationships between them. These relationships transform into dependencies between the Ada packages.

Once each of the objects is defined, the operations performed by or performed upon each object are identified and allocated to the Ada packages. The Ada packages give these operations a "home". This is where the operations will reside within the software system.

Using Ada along with a graphical representation is the best way, I think, to describe the static part of the preliminary design of a software system. However, a representation based strictly on packages and their dependencies does not account for all aspects of the software.

A software system, no matter how it is decomposed, is still either a single sequential flow of control or multiple sequential flows of control between the operations. This is the dynamic part of the preliminary design. The understanding of these flow(s) of control is essential to the review process and maintenance process.

DOD-STD-2167A, specifically Sections 3 of DID-MCCR-80012A (the Data Item Description (DID) for the Software Design Document (SDD)), is written with a bias toward the dynamic representation (the process oriented representation) of the design. Choosing a CSC definition that does not fit into a process-orientation is like fitting a square peg into a round hole; it is difficult and frustrating work.

Problems with Commonly used Definitions

Many projects rationalize liberal interpretations of the requirements in the SDD DID to make their definitions fit. However, they forget that how CSCs are defined effects more than the SDD. The definition effects other parts of the software development life cycle and other software functions (e.g. CSC Integration and Test, Software Configuration Management). The CSC definition must provide consistency across all parts of development.

For example, many projects define a CSC to be an Ada package. An Ada package is a collection of resources, is not executable and therefore has no process-orientation associated with it. This definition has to be forced to fit the documentation requirements of the SDD DID. Following are some liberal interpretations of some Section 3 requirements based on a package being a CSC:

1. Paragraph 3.1.1 of the SDD (CSCI Architecture) describes the internal organization of the CSCI. This description includes a description of the relationships among the CSCs by identifying and stating the purpose of each CSC-to-CSC interface. This interface has to summarize the data transmitted via the interface. An interpretation used for the CSC-to-CSC interface is the dependency between packages. This is a very liberal interpretation because dependency does not provide for data being passed and never will. A process oriented definition of a CSC is better suited to fit the requirements of this paragraph.

2. Paragraph 3.1.3 documents the memory and processing time allocation of each CSC. An interpretation often used is to allocate memory and processing time to each executable resource in the package and assign the sum of the individual allocations as the allocation for the package. This liberal interpretation yields meaningless data, especially for packages that are a product of object-oriented design. In an object-oriented design, each of the operations in a package are not meant to be executed in sequence, might not be part of the same CSCI or might not be called at all in this software system. To reviewers and maintainers of the CSCI this data is useless.

3. Paragraph 3.2X(b) asks for the preliminary design of this CSC to be described in terms of execution control and data flow. An interpretation that has been used on a real project to fulfill the requirements of this paragraph is to describe the parameters of each subprogram (or task entry) in the package and describe the conditions under which each are called. Again, this is a liberal interpretation of the DID requirements that provides a disjoint flow. The DID is asking for something that is more process oriented, something flows from one part of the CSC to another.

These interpretations are forcing a package to be something it is not. If this definition is allowed to be used and these interpretations are accepted as fulfilling the documentation requirements then other software development activities are rendered meaningless. For example, Integration and Test of a CSC means composing the CSC from its component parts and verifying that the CSC works as a whole. The parts (subprograms?) of a package that was designed using an object-oriented methodology are not meant to work in concert with each other. There is nothing to integrate. The meaninglessness is a direct result of a faulty CSC definition.

An Alternative Definition

A definition of a CSC that is process oriented, that fits well with 2167A's documentation requirements and software testing requirements and provides a consistency across the development process is a thread of control.

A thread of control is defined as the execution path that begins with a stimulus and ends with a response to that stimulus. A stimulus consists of one or more inputs plus any conditional qualifiers; a response consists of one or more outputs plus conditional qualifiers generated as result of the input event. An entire software system may be decomposed into a set of stimulus/response elements.¹

Figure 1 portrays an example of the derivation of a thread from requirements in a requirements specification. The figure shows that the thread, numbered 109 and named New Hostile Track, is derived from two software requirements and receives input from and provides output to the external interfaces. The stimuli for the thread is put in the left rectangle and the responses for the thread are put in the right rectangle. The requirement numbers from which the thread is derived are listed below the graphical representation of the thread.²

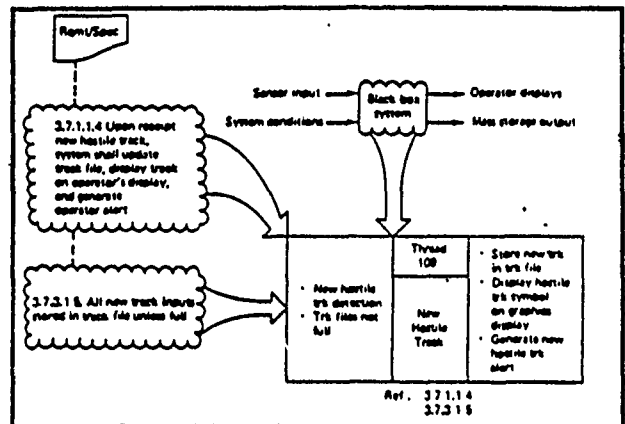


Figure 1. Thread Derivation

The entire software requirements specification can be represented by a set of threads (see figure 1) which are logically connected to each other by arrows which denote sequence. Quite often a conditional qualifier associated with the input event of a thread is the successful completion of the function represented by the previous thread in the sequence.¹

Using the Proposed Definition

So, the preliminary design phase can be summarized by the following steps (this assumes an object-oriented methodology):

1. Derive the threads (stimulus/response elements) from the software requirements specification. This is the dynamic model of the preliminary design. Graphically represent the sequential flow of the threads.
2. Keeping the threads in mind, use object-oriented design to decompose the software system into objects. Graphically represent the Ada preliminary design using a graphical notation such as Booch, Buhr or Texel's methodology. Transform the graphical representation into compilable Ada. This is the static model of the preliminary design.

3. Map each thread from the dynamic model into the static model of the preliminary design. This will identify the specific software modules which, when executed, perform the function of the thread.¹

4. Document, in Section 3 of the SDD, the dynamic model (the threads and their interfaces). This documentation should be done in terms of the software modules defined in the static model. Include the graphical representation as reference. Document the static model in Section 8, Notes, of the SDD. Include both the graphical representation and the Ada representation.

Following is how to write Sections 3.1.1, 3.1.3, and 3.2.X(b) of the SDD using the proposed definition of a CSC. Note how using a CSC definition that is process oriented makes these sections more meaningful.

1. Paragraph 3.1.1 describes the internal organization of the CSC partly in terms of the CSC-to-CSC interfaces. For each interface, the data transmitted via this interface is identified. With the proposed definition the graphical representation of the dynamic model displays each interface between the threads as a data flow. The data transmitted via this interface can then be determined from the static model.

2. Paragraph 3.1.3 describes the memory and processing time allocation of each CSC. All of a thread, from beginning to end, will be executed at some point in the system software. An allocation of memory and processing time (either worst case or different allocations for various paths) that represents a sequential flow of processing can be made to each thread. These allocations, along with the graphical representation of the dynamic model, can be used to provide timing and sizing estimates when executing the software under varying conditions.

3. Paragraph 3.2.X(b) describes the CSC in terms of execution control and data flow. The process-oriented characteristics of a thread fits naturally with what 2167A is looking for in this section. The entire thread can easily be described in terms of execution control and data flow because that is basically the definition of a thread. Also, because the static model was also produced, and the threads mapped to it, the execution control and data flow can be described in terms of the real names of subprograms, task entries and data in the static mode. The option of defining sub-level CSCs also fits with the proposed definition of a CSC. If a thread is too large, it may be decomposed into children threads and each

of these may be described separately. Sublevel CSCs may even be used for different paths within a small thread.

A by-product of the proposed definition is that requirements allocation to the CSC level is already accomplished for you when the threads are identified. Deriving the threads from the requirements specification itself simplifies the requirements allocation. When a thread is identified from the requirements specification, all associated requirements can be allocated at that time.

Advantages of the Proposed Definition

The advantages of defining a CSC as a thread of control are described below:

1. If a CSC is defined to be an Ada package (or some other Ada entity), massive changes to Section 3 of the SDD will be needed during detailed design. It is impossible to define every Ada package during preliminary design. Each package added during the detailed design phase must be documented in Section 3. A rewrite of the interfaces of many CSCs is inevitable. Defining a CSC as a thread of control reduces the probability of changes to sections of the SDD that have already been completed. Threads are identified from the requirements specification and should all be known during preliminary design.

2. The proposed definition enhances the meaningfulness of CSC Integration and Test and provides a smooth transition into that phase of the software life cycle. The way CSC Integration and Test is described in 2167A, the only meaningful definition of a CSC is a process-oriented one. The component parts of the thread (its CSUs), which are already tested in the previous phase of the life cycle, can now be verified as an entity (the entire thread). As described earlier, a package definition does not work well.

3. Threads of control as a CSC fits naturally with the idea of builds. Many real-time systems verify their software through builds. Builds are composed of threads and incrementally demonstrate a significant partial functional capability of the system.¹ As CSCs composing a build are integrated and tested the build can be integrated by composing the threads. The idea of builds would suffer if a CSC was defined as an Ada package. Many more CSCs, many with parts which would not be applicable to a build, would need to pass CSC Integration and Test just to get the correct functionality of the build.

Disadvantages of the Proposed Definition

The disadvantages of defining a CSC as a thread of control are described below:

1. The proposed definition does not map directly to any Ada program unit or set of program units. When designing software in Ada it is easier to describe the software in terms of Ada program units. However, as described previously, the static model of the design is in Ada and this model can be documented as applicable in Section 8 of the SDD (Notes).

2. Using the proposed definition of a CSC, Section 3 of the SDD does not exhibit any Ada architecture. Section 3 would document the dynamic model of the design. The static model (the Ada architecture) is an important part of the documentation and should be documented. This documentation, prepared in contractor format, would be included in Section 8 of the SDD.

Conclusion

The preliminary design of a software system has two parts, a static model and a dynamic model. Both of these models are important to the review and maintenance of a software system. The dynamic model, however, is the only one which fits naturally with the requirements of DOD-STD-2167A. The dynamic model is the one that should be documented in Section 3 of the SDD.

The dynamic model is process-oriented. Therefore, a CSC should be defined to be process-oriented. Threads of control

are process-oriented and can be used to represent the dynamic model. Threads fit naturally with the requirements of 2167A and provide a smooth transition into other software development activities. Therefore, a CSC should be defined to be a thread of control.

However, this definition does not preclude the use of object-oriented methodologies. The static model can be represented with Ada that is the direct result of an OOD. This model should also be documented in the SDD, but informally in the notes section.

Biographical Sketch

John K. Cannella, P.P. Texel & Company, Victoria Plaza, Building 4, Suite 9, 615 Hope Rd., Eatontown, NJ 07724. Mr. Cannella is the Director of Software Development at Texel. Mr. Cannella, as a consultant, has been technical lead on numerous DOD-STD-2167 and DOD-STD-2167A software development efforts. These efforts include the Small ICBM Terminal Controller, the MV-22 Operational Flight Trainer and the Radar System Improvement Program.

1. Deutsch, Michael S.: Software Verification and Validation, Prentice Hall, Inc., 1982

2. Carey, Robert and Bendic, Mark, "The Control of a Software Test Process," Proceedings Computer Software and Applications Conference 1977 (New York: IEEE), IEEE Catalog No. 77CH1291-4C.

TEACHING ADA: LESSONS LEARNED

David Naiditch

Hughes, Radar Systems Group
Los Angeles, California

Abstract -- This paper is based on lessons I've learned from teaching Ada programming courses over a period of 4 years at Hughes; at the University of California, Los Angeles (UCLA) extension; and at various Ada seminars. The courses have ranged from brief, half-day overviews to 20-week training sessions. The students in these courses had varying backgrounds: some were well versed in many high-level languages and programming principles; at the other extreme, some were learning Ada as their first high-level language.

Index Terms -- Ada, education, training, software engineering principles, exercises, textbooks, preconceptions about Ada

1. INTRODUCTION

This paper is targeted for Ada educators, especially those who teach introductory Ada courses. The teaching techniques I present in this paper have been successfully used in many Ada courses that I have taught over the years. How well a teaching technique works depends to some extent on the instructor's style and temperament. I, therefore, realize that some techniques that work well for me might not be useful to others. I hope, however, that educators will find some of my ideas useful and worth pursuing.

2. INSTRUCTION METHODS THAT DON'T WORK

When I first starting teaching Ada about 4 years ago, my courses were rather abstract. I left out many details of Ada syntax and semantics, hoping that students would learn them on their own, and concentrated instead on the grand software

engineering principles that the Ada language so effectively supports. This approach seemed reasonable to me because it was embraced by many Ada books and appeared to stress what was most important about Ada. However, I soon discovered that students were not learning the details of Ada well enough to write programs without clutching their worn-out Ada manuals. Questions asked in class often arose from the frustration students felt in dealing with the many subtle and complex nuances of Ada. For example, students would ask why one cannot write for loops such as

```
for INDEX in -1 .. 10 loop ...
```

or why array aggregates sometimes need to be qualified, or why record discriminants cannot be used in an expression, or why anonymous arrays within the same declaration are type incompatible, or why a procedure cannot read its own out parameters, or why type conversion must be applied to the product of two fixed point numbers, and on and on. As I pontificated about the virtues of adhering to proper software engineering principles, students would nod in agreement and urge me to finish so that they could get back to the practical matter of getting their code to compile and then execute without raising the dreaded `Constraint_Error`.

Part of the problem with not covering the details of Ada was the lack of Ada books that were simple and complete enough for students to learn the details of Ada on their own. Books were either too

advanced or too superficial for beginning students. Furthermore, books stressing software engineering principles didn't provide enough information about the Ada language itself. Such books frustrated many students by illustrating software principles with code examples that students could not yet fully understand. And in almost all cases, the books were poorly written. The complexity of the Ada language remained obscured by the complexity of the English language used to write the book. As a result, too much class time was taken up with my explanations of what the authors were attempting to say.

Another problem with my original teaching approach was that students didn't fully absorb what I was saying. When presented in abstract terms, software engineering principles come off like motherhood and apple pie. Everyone supports them, but so what! Students thought that programmers using common sense follow software engineering principles as a matter of course. When reviewing student programs, however, I realized that many of these principles did not get put to practice. Package specifications contained information that should have been hidden in package bodies. Types that should have been private or limited private were often public. Global variables were often used without justification. Literals were frequently employed instead of constants or attributes.

3. INSTRUCTION METHODS THAT WORK

3.1 Be Concrete

I found that the coding style of students dramatically improved when I made my lectures less abstract. Instead of ignoring many of the details of Ada, I now present them in a methodical and organized manner. Instead of discussing software engineering principles in abstract terms, I give them concrete form in numerous code examples and exercises. In addition, I discuss a software engineering principle in detail only when students are ready to learn about the Ada constructs that

support or enforce that principle. For example, I don't spend too much time discussing information hiding until students know the "Pascal subset" of Ada and are prepared to tackle packages. With this teaching approach, not only do software engineering principles become easier to put into practice, but the details of Ada appear less complex and arbitrary. In other words, teaching Ada in the light of proper software engineering principles simplifies Ada by uniting seemingly unrelated rules and by providing a rationale for many Ada features that may otherwise seem arbitrary.

3.2 Use a Self-Explanatory and Comprehensive Ada Textbook

In order not to get too bogged down explaining all the gory details of Ada syntax and semantics, I wanted an Ada book that covers such details clearly enough for students to learn on their own. I was hoping that using such a book would free me to spend time on higher-level issues. When I began teaching 4 years ago, I could not find such a textbook and in frustration wrote my own, *Rendezvous with Ada: A Programmer's Introduction*, which John Wiley & Sons published in 1989. Today, however, there are a number of good books to choose from, mine, of course, being the best.

3.3 Use Humor

Another teaching technique that I have found very helpful is the use of humor. Humor can make dry material more palatable. Here is an example of an If statement. (All examples are taken from my Ada book.)

```
if ART = ABSTRACT or ART = NONOBJECTIVE
then
    PUT_LINE ("Whine and complain");
    PUT_LINE ("Leave room");
else
    PUT_LINE ("Stay and enjoy");
end if;
```


Or consider this **case** statement that would please any member of the Audubon Society.

```
case FLOCK_OF_BIRDS is
  when LARKS           => PUT_LINE
    ("An exultation of larks");
  when PEACOCKS        => PUT_LINE
    ("An ostentation of peacocks");
  when CROWS           => PUT_LINE
    ("A murder of crows");
  when GEESE           => PUT_LINE
    ("A gaggle of geese");
  when SPARROWS        => PUT_LINE
    ("A host of sparrows");
  when MAGPIES         => PUT_LINE
    ("A tiding of magpies");
  when PHEASANTS       => PUT_LINE
    ("A bouquet of pheasants");
  when OWLS            => PUT_LINE
    ("A parliament of owls");
  when STARLINGS       => PUT_LINE
    ("A murmuration of starlings");
  when PARTRIDGES      => PUT_LINE
    ("A covey of partridges");
  when NIGHTINGALES    => PUT_LINE
    ("A watch of nightingales");
  when WOODPECKERS     => PUT_LINE
    ("A descent of woodpeckers");
  when others          => PUT_LINE
    ("No special name");
end case;
```

If you need a coding example where text is written to a file, resist the easy way out: stuffing the file with boring information such as a person's name and address. Instead, dare to be silly and, for instance, fill the file with words of wisdom we all hope to find in our fortune cookies.

```
PUT_LINE (WISDOM, "Don't let your karma run"
  & " over your dogma");
PUT_LINE (WISDOM, "You can lead a horse to"
  & " water, but a pencil must be lead");
```

```
PUT_LINE (WISDOM, "Time flies like an arrow,"
  & " but fruit flies like bananas");
PUT_LINE (WISDOM, "Every bird can build a"
  & " nest, but not everyone can lay an egg");
PUT_LINE (WISDOM, "When there's a will,"
  & " there's a won't");
```

As a final example of humor in code, here are some entertaining ways to illustrate string declarations:

```
MY_CAT: STRING (1..13) := "Meow Tse-tung";
MY_BOA: STRING (1..15) := "Julius Squeezer";
MY_COBRA: STRING (1..11) := "Herman Hiss";
MY_NEWT: STRING (1..16) := "Sir Isaac Newton";
```

Examples such these make learning more fun without being too distracting. Many students have thanked me for adding humor to my courses. Students who don't find my code amusing never seem to be bothered (at least, not visibly so).

3.4 Use Interesting Examples

In addition to humor, material can be made more palatable by using interesting examples. For instance, I often ask students to write an Ada program that illustrates the "twin paradox" that is a consequence of Einstein's Theory of Special Relativity. Consider two twins, where one gets into a spaceship and travels near the speed of light, while the other stays on Earth. Upon returning to Earth, the traveler will be younger than the twin who stayed behind. This program is based on the simple formula:

$$A := T * \text{SQUARE_ROOT} (1.0 - (P/100.0) ** 2);$$

where

A is the number of years that the traveler will age

P is the percent of the speed of light that the traveler will be traveling

T is number of earth years that the trip takes

After writing this program, students enjoy plugging in different values for T and P to determine how little the traveler will age compared to the time that has elapsed on Earth.

Or consider an example of a **while** loop that illustrates Ullam's conjecture:

```
-- N is a positive integer
-- EVEN is a Boolean function
-- that determines if N is even
while N /= 1 loop
  if EVEN (N) then
    N := N / 2;
  else -- odd number
    N := 3 * N + 1;
  end if;
  PUT (ITEM => N, WIDTH => 6);
end loop;
```

This **while** loop generates a series of numbers that terminates with 1. For example, if N is 7, then the following series is generated:

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

Computers have worked this problem with many different positive numbers. Without exception, the number 1 is eventually reached. However, no one has yet been able to prove mathematically that the number 1 will always be reached. (By the way, some large numbers, like 341, require only 11 steps to reach 1, whereas some small numbers, like 27, require 111 steps.)

3.5 Overcome Negative Preconceptions about Ada

Students sometimes enter my class harboring negative preconceptions about Ada. The negative preconceptions take different forms. Some students fear that Ada is such a large and complex language that they have to be a language lawyer to use it correctly. On the opposite end of the spectrum, those who have been FORTRAN programmers for

20 years or more might say "Ada schmada, it's just another programming language, and I already know how to program." Another preconception is that Ada is too inefficient for real-time programming. And then there is my favorite preconception--that Ada is too paternalistic. Most of the other preconceptions go away as the class progresses, but not this one. This preconception is by far the most difficult to overcome because it is tied to one's political philosophy. Just as libertarian-minded motorcyclists resent being forced to use safety helmets, so do libertarian-minded programmers resent being forced to use safe programming practices. Typically, I hear, "Why won't Ada let me do this?" Such programmers want the freedom to take risks. They don't want the language to restrict them from writing code that is error prone, unreliable, or difficult to understand. Such libertarian sentiments are frequently expressed by C programmers who enjoy hacking.

I find that the best way to deal with this last preconception is to explain the programming domain for which Ada was designed. I discuss how Ada was developed in response to the software crisis. I point out that Ada's emphasis on reliability and maintainability is especially relevant to large programs that are long lived and frequently modified. I show code that illustrates how Ada's restrictions effectively support reliability and maintainability. I admit, however, that such emphasis is less relevant for small, short-lived programs that do not need to be very reliable--programs developed by home hobbyists, for example.

3.6 Have Students Modify Large Programs

When learning a new language, it is important to write at least one large program. This is especially important with Ada since Ada is tailored to large program development. A major problem with teaching Ada is that not enough time is usually given for students to learn all of Ada's features, much less to develop even one large program. I'm typically

asked to teach classes lasting under 40 hours. To write large programs, students need to know about such features as scalar and composite types, subtypes and derived types, attributes, control structures, subprograms, packages, and program structure. By the time this material is adequately covered, the course could be half over. In the remaining half of the course, there isn't enough time for students to write large Ada programs while also learning about Ada's more advanced features, such as generics, exception handling, and tasking.

Despite such time constraints, because large program development is important, some instructors still require a large program to be written. Not surprisingly, the resulting program is usually terribly written. Writing bad programs may be worse than not writing them at all; it may only instill bad habits that are later difficult to unlearn.

As an alternative to having students write large programs, I sometimes have students modify large programs that are already written. These large programs should be well written and well documented. I ask students to make modifications that require them to carefully study the program. Instead of reinforcing bad coding techniques, students will be examining code that exemplifies good coding techniques.

So how can instructors find large Ada programs that are well written and well documented? If you don't have such programs on hand, try going through the public Ada repositories. Coding gems can sometimes be found lurking among all the mediocrity.

3.7 Initiate Code Exchanges

Students often lack objectivity when evaluating their own code. They often have an easier time seeing problems with other people's code. I sometimes have students develop a program that they consider well written. Everyone in the class then exchanges programs. Students are often

surprised when others in the class cannot understand their code.

3.8 Give Classroom Exercises

Another teaching technique I have found effective is to break up long lectures with classroom exercises and problems. Even if Ada compilers are not available in the lecture room, exercises are nevertheless effective in reinforcing concepts after they have been presented. Exercises might include critiquing code fragments, finding coding errors, and figuring out what a code segment will output.

3.9 Provide Hands-On Training

Learning to effectively program in Ada requires a lot of hands-on experience. Ideally, the training should be tailored to the kind of programming that students are expected to do. For example, students who program for embedded real-time systems need to learn a lot about the low-level features of Ada that allow programmers to get down to the "bare silicon" of the machine. In addition, real-time programmers must be aware of the trade-offs between performance and memory. For example, when should `pragma inline` be used? If a compiler offers the choice, should generics use code expansion or code sharing? Real-time programmers must also be aware of trade-offs between code understandability and performance. For example, should the Ada rendezvous be used to make code more understandable or is there too great of a performance hit. Similarly, what about using functions that return an unconstrained array type? What about using unconstrained discriminated records? And then there is the issue of whether runtime checking should be suppressed. If it should be suppressed, should it be globally suppressed or only suppressed within a given scope or for certain kinds of checking? Furthermore, does the use of allocators or other features that require dynamic memory allocation need to be restricted because of

limited heap space?

These issues are very important for resource-critical projects such as real-time embedded applications. Other sorts of applications, however, may have very different concerns.

4. SUMMARY

When I first started teaching Ada about 4 years ago, my lectures were too abstract. Instead of covering the details of Ada syntax and semantics, I concentrated on teaching about the sound software engineering principles that Ada supports. I hoped that students could learn about the details of Ada from reading their Ada textbooks. Unfortunately, I could not find beginning Ada textbooks that were well written and comprehensive enough to free me from having to spend time explaining the subtle and complex features of Ada syntax and semantics.

Another problem with my original teaching approach was that students didn't understand my discussion of software engineering principles well enough to put the principles into practice. Their code was often written as if software engineering principles were never mentioned.

I now get much better teaching results by being less abstract. Instead of ignoring many of the details of Ada, I present them in a methodical and organized manner. Instead of teaching software engineering principles in abstract terms, I give them concrete form in numerous code examples and exercises. Furthermore, I delay discussing a software engineering principle in detail until students are ready to learn about the Ada constructs that support the principle. Finally, by tying software engineering principles to particular Ada features, Ada appears less complex, more unified, and hence easier to learn.

Over the years of teaching Ada, I have incorporated other useful teaching techniques. I use humorous and interesting code examples to help make material more entertaining and therefore

easier to learn. I successfully overcome some negative preconceptions about Ada that many students harbor. Finally, I give students assignments that I have found to be particularly beneficial. These assignments include modifying large programs, participating in code exchanges, working on classroom exercises, and engaging in hands-on training geared to particular applications.

David Naiditch has been working at Hughes for over 9 years, where he is currently an Ada project engineer. Mr. Naiditch has over 15 years' experience in the instruction of high-level programming languages. For over 4 years, he has been teaching a wide variety of Ada courses at Hughes; at the University of California, Los Angeles (UCLA) extension; and at seminars across the country. Mr. Naiditch is the author of the book *Rendezvous with Ada: A Programmer's Introduction*, published by John Wiley & Sons in 1989.

Mailing address: Hughes, David Naiditch (Bldg. R1, MS 6A27), Project Engineer, Processor Division, Radar Systems Group, P. O. Box 92426, Los Angeles, CA 90009.

AN ISSUE TO BE CONSIDERED WHEN REVISING DOD-STD-2167A

Russell J. Abbott

Department of Mathematics and Computer Science
California State University, Los Angeles

Abstract. This note discusses a problem that needs to be addressed when considering revisions to DOD-STD-2167A. Although the problem is understood most easily when software is considered from an object-oriented perspective, it applies to software of all kinds. The problem derives from the fact that software has two distinct structures:

- a static structure, the structure of the source code;
- a dynamic structure, the structure of the software in operation.

These structures are frequently quite different, and it is important that they each be specified and documented carefully.

As written, DOD-STD-2167A does not recognize that these structures are distinct. It forces information about both structures to be interwoven in a way that leads to complications and difficulties that would not exist were the two structures considered separately.

1. Introduction

In much of traditional software engineering, little or no effort is made to distinguish between the structure of software as source code (i.e., how the software source code is organized) and the structure of the operational elements derived from the software (i.e., the structure of the sys-

tem in operation). In fact, in much of software engineering it has been implicitly assumed that these structures are identical.

In object-oriented software, the distinction between these two structures is striking. Consider a system that includes two queues that play distinct roles in an operational system. For example, one queue may hold messages to be processed and the other jobs to be done.

In an object-oriented implementation, these two queues would be distinct objects which are instantiated from a single `queue` object description.*

- Both operational queues would appear in a description of the functional design of the system, and their two distinct roles in that design would be discussed.
- Since there is only one `queue` object description, it would appear only once in the system's source code. It would also appear once in the list of work assignments.

In object-oriented software, the software product, i.e., the source code, consists almost entirely of object descriptions. The operational elements, the objects, do not exist until the system is put into operation. This leads to a very sharp distinc-

* We use the term *object description* instead of *class* since *object description* is not tied to any particular programming language.

tion between an object-oriented system's operational structure (the interrelationships among the objects) and the structure of its source code.

Section 2 explains very briefly how this distinction arose. Sections 3 and 4 discuss paragraphs within DOD-STD-2167A³ and DI-MCCR-80012A⁴ for which this distinction is relevant. Section 5 contains a summary and conclusions.

2. Objects and Object Descriptions

Object-oriented programming grew out of work on data types (and especially abstract data types) in programming languages. The notion of *object description* in object-oriented systems is an extension of the notion of (*abstract*) *data type* in traditional systems. Just as a *data value* in a traditional programming language is an instance of a data type, an *object* in an object-oriented system is an instance of an object description.

The primary difference between object-oriented systems and traditional abstract data type systems is that in object-oriented systems, the objects (i.e., the values) are not static and passive. Instead they have a built-in ability to perform the operations that in a traditional language would be defined as the abstract data type operations.

For example, in an object-oriented system there may be an object description that plays the same role that the type counter plays in traditional systems. During operation, there may be a counter object that contains a 3. When presented with a request to increment itself, such an object will respond by adding 1 to its internal value making it a 4. In simplest terms, then, an object is a data element that is capable of accepting and responding to requests that it perform operations.

It is important to realize that objects may be major system elements as well as minor components like counters and queues. An object-oriented system for communicating with satellites, for example, may include objects that track the satellites. There

may be a separate tracking object for each satellite being tracked.

2.1 The Objects are not Delivered

In object-oriented software, the objects are the operational elements, i.e., the elements that do the work when the software operates. An object-oriented system in operation generally consists of (nothing but) a (usually large) collection of objects interacting with each other and with the outside world. Thus it is the objects that do whatever work the software does as it operates. The design of the system is expressed in terms of objects and their interactions.

Yet objects do not exist prior to system operation. *An object is instantiated (i.e., comes into existence) when storage is allocated to it.* Furthermore, most objects do not persist during the entire time the system is in operation. They come into existence and pass from existence as the system operates. *An object passes from existence when its storage is deallocated.* The collection of objects in existence at any particular time is generally quite dynamic.

There are a number of corollaries to this distinction between objects and object descriptions.

- The design of a software system, i.e., the description of how the system works, is best understood in terms of its objects, not its object descriptions.
- The components that make up the system, i.e., the object, and the system itself will *not* have been created and are *not* delivered when the software developers finish their work. This is very different from most system acquisitions in which the system *is* delivered by the developer.

An object-oriented software developer produces and delivers only the object descriptions from which the system components, i.e., the objects, will be instantiated while the system is in operation.

- The task of producing the object descriptions, i.e., the software deliverables, is organized primarily in terms of the object descriptions, not the objects. So the organization of development tasks does not necessarily parallel the organization of the system being developed.
- Objects cannot be placed under configuration control; only object descriptions and system design documents can be placed under configuration control.

It should be clear from the preceding discussion (and it will become even clearer as we proceed) that the system's *operational* structure (rather than the structure of its source code) is the one and only structure that determines whether a software system will operate as desired. A system's design is a description of and rationale for its operational structure. The fundamental issues are:

1. how a system's operational structure will be identified;
2. how a system's operational structure will be related to the structure of the system's source code.

2.2 Objects and Object Descriptions in Non-Object-Oriented Software

The distinction between objects and object descriptions applies to software that was not written in an object-oriented programming language as well as it does to software that was written in object-oriented languages. The most relevant example is Ada².

Although Ada has object-oriented features, it is not an object-oriented programming language in the sense discussed above. No general means are provided to instantiate objects that have operation definitions built into them.

In Ada, the term *object* refers to a unit of storage that may hold values. As indicated above, this definition is subsumed by the object-oriented definition of *object* which adds to it that an object also

has the ability to respond to requests that it perform operations.

There is, however, an Ada construct that is quite object-like in the object-oriented sense. Ada task types are very similar to object descriptions in object-oriented languages, and tasks instantiated from task types are quite similar to objects.

- Both objects and instantiated tasks have storage allocated to them for internal use.
- Both objects and instantiated tasks have means to respond to requests that they perform operations.
- Both objects and instantiated tasks come into existence and go out of existence while the system is in operation. Neither exists before the system is put into operation.
- There may be many more objects and instantiated tasks than objects descriptions and task types.
- The design of a system that uses objects and instantiated tasks is better understood in terms of the relationship among those objects and tasks than in terms of the relationship among the tasks types and object descriptions.

Other than the task type, however, Ada is not an object-oriented language. In Ada, as in most of the programming languages from which it was derived, the distinction between objects and object descriptions is not made precise. In some cases, there is very little difference.

This is most easily seen in the earliest programming languages. In languages such as assembly language and Fortran, the structure of the source code, is identical to the structure of the operational system. In these languages, each software module is tied to some fixed locations in storage. A single module of software source code cannot give rise to multiple objects.

The distinction between objects and object descriptions first arose with re-entrant subprograms and the use of a stack for storing local data. Once

the data used by a software component could be separated from the component's code, components could have many distinct incarnations. Each instance of the local storage for such a component corresponds to what in object-oriented terminology is now called an object, and the subprogram code corresponds to what we are calling an object description.

However, such early objects were instantiated in lock step with the call and return of the subprograms for which they provided local storage. Consequently, subprogram object descriptions could not give rise to multiple objects that existed simultaneously. The objects instantiated from a subprogram object description came into existence and went out of existence in strict sequence. So even though the so called *fan-in* problem of utility routines has been creating conceptual software organizational difficulties for years, it has been possible to ignore the problem without grave damage.

Now, however, with the increasingly widespread use of parallel and distributed processing and with true object-oriented programming languages, there is virtually no choice but to understand large software systems in terms of interacting processes and objects. As a result, the distinction between the structure of the source code and the structure of the operational system can no longer be ignored.

Even in programming languages in which object-like constructs are not directly available, programmers find themselves forced build comparable structures. They do so either through the use of operating system services such as interacting processes or by implementing mini-object-oriented systems in global common storage.

2.3 Object-Oriented Languages Provide No Way to Specify System Architecture

Although object-oriented languages now provide means to specify objects, a significant weakness of most such languages is that they provide no way to specify system architectures.

The primary focus of most object oriented languages is on specifying object descriptions. Other than the actual code that causes objects to be instantiated (which is typically sequential and at a very low operational level), very little effort, if any, is given to specifying how objects, once instantiated, will be organized. In particular, object-oriented programming languages generally provide no declarative means for describing object structures and interactions.

Ada's somewhat old-fashioned, non-object-oriented nature is an advantage here. The Ada package construct is truly a system architecture construct. An Ada package is an object, not an object description. As such, a package cannot be instantiated multiple times; it already is instantiated. The nesting of package source code components defines a system architecture.

Yet even the package construct is not a pure system architecture mechanism. A package may be declared within a subprogram or a task type. Each call to such a subprogram and each instantiation of such a task type will lead to a new instantiation of the embedded package. In this sense the package is not a pure object since the source code for a package may lead to multiple package instances.

3. DOD-STD-2167A Software Development

This section discusses paragraphs in the current DOD-STD-2167A for which the distinction between objects and object descriptions is relevant. It is divided into subsections that discuss respectively sections 3, 4, and 5 of DOD-STD-2167A.

3.1 Section 3. Definitions

Section 3 of DOD-STD-2167A is a list of definitions. The two definitions of interest to us are the definitions of CSC and CSU.

Paragraph 3.8 Computer Software Component (CSC) and 3.11 Computer Software Unit (CSU). DOD-STD-2167A defines *Computer Software Component (CSC)* and *Computer Software Unit (CSU)*.

The primary intent of these definitions is presumably to provide a framework for software testing and integration. The CSUs are intended to be testable atomic elements; the CSCs are intended to be testable aggregated elements.

The problem is that the terms *CSC* and *CSU* are used to refer both to the structure of the source code as well as to the structure of the operational system. In some places, CSCs and CSUs are understood to be work products, i.e., software source code; in others, they are understood to be system components.

As work products, CSCs and CSUs are object descriptions and collections of object descriptions. As system components, CSCs and CSUs are objects and collections of objects. As we indicated above, both kinds of structures are important. The problem is that they are distinct and need to be dealt with separately.

To avoid confusion we will not use the terms *CSU* and *CSC* except when referring to the text in the standard. Instead, we will continue to use the terms *object* and *object description*.

3.2 Section 4. General Requirements

This section discusses requirements imposed by Section 4 General Requirements.

Paragraph 4.1.1. Software Development Process. This paragraph requires the software developer to code and test Computer Software Units (CSUs). An immediate terminological problem arises because *object descriptions* are coded, but *objects* are tested.

Terminological problems are usually easy to fix. *Testing an object description* can be understood to mean testing objects instantiated from that object description and *coding an object* can be understood to mean coding the object description from

which the object is to be instantiated. From here on, no further mention will be made of problems that can be resolved by simple terminological conventions.

Paragraph 4.2.5 Computer software organization. This paragraph requires the contractor to decompose and partition each CSCI into CSCs and CSUs. The associated Figure 3 illustrates a possible decomposition.

The problem here is that there are a number of ways to decompose a system, and the paragraph is not specific about which one is desired.

- **Functional.** A functional decomposition is one that shows how a system is composed of subelements whose functions, when combined, achieve the system objectives. A functional decomposition of an object-oriented system would show what the objects are and how they interact. A functional decomposition would tend to illustrate dynamic interactions. (See Abbott¹ for a discussion of how functional decomposition, properly understood, is quite compatible with object-oriented software.)
- **Parts List.** A parts list decomposition is one that shows the component parts for each component. A parts list decomposition is a tree structured, hierarchical decomposition of a system down to its atomic elements. Such a decomposition includes every operational component that may exist during the life of the system without regard to the time during which the components exist.
- **Work Breakdown.** A work breakdown decomposition is one that shows how the system is to be developed in terms of the required work tasks and the deliverables.

To understand the differences among these three kinds of decompositions, consider again a system with two queues.

- In a functional decomposition both queues would appear in the decomposition, their different roles would be identified, and their in-

teractions would be illustrated as in a dataflow diagram.

- In a parts list decomposition, both queues would be shown in the decomposition, but there would be no indication of whether or not they interact.
- In a work breakdown decomposition, only the single queue object description would appear, and it would appear only once.

All three kinds of decomposition are useful, but it is important to recognize the differences among them. Both functional and parts list decompositions identify system objects. In addition, a functional decomposition shows the dynamic interrelationships among the objects. Although a parts list decomposition includes each operational element, unlike a functional decomposition, it does not indicate (a) whether pairs of components interact, (b) how interactions occur, or even (c) whether components exist simultaneously. In fact, in many large systems, pairs of components do not operate simultaneously. For example, in a satellite system, a launch subsystem and a payload data analysis subsystem may not exist simultaneously. Yet both would appear in the operational decomposition.

In contrast, a work breakdown decomposition does not identify the system objects at all. Instead, it shows the interrelationships among the object descriptions.

Rather than try to divine which kind of decomposition is intended by the current standard, it would seem more sensible to realize that one will probably want all three kinds of decomposition at various times during system development.

Paragraph 4.5.1 Configuration identification. The contractor is required to identify each CSC and CSU and the version of each CSC and CSU to which the corresponding software documentation applies.

Both (a) the designs of objects and higher level components and (b) object descriptions can be kept under configuration control. The actual objects cannot be kept under configuration control

since they exist only within the computer and only when the system is in operation.

3.3 Section 5. Detailed Requirements

This section discusses paragraphs from section 5 of DOD-STD-2167A.

Paragraphs 5.3.2 and 5.4.2 Software Engineering (within 5.3 and 5.4 Preliminary and Detailed Design respectively). In sections 5.3.2.1 and 5.4.2.1 the developer is required to establish design requirements for each CSC and CSU.

Objects and higher level components have design requirements. Object descriptions do not have design requirements; they are designs since they describe how the objects they describe will operate.

4. DI-MCCR-80012A Software Design Document

DI-MCCR-80012A specifies the form and content of a Software Design Document (SDD). According to DI-MCCR-80012A, an SDD should describe a CSCI as composed of CSCs and CSUs. It should also describe the functions of the CSCs and CSUs and the relationships among them.

Again, the problem is here is that the existence of two distinct structures is not recognized. Depending on one's perspective, a CSCI is either (or both) source code and an operational system. When one is concerned about how a CSCI will operate and whether it will produce the desired results, one is concerned with its operational structure. When one is concerned about how the source code is organized and how the work required to produce that code is broken down, one is concerned with the organization of the object descriptions.

Paragraph 10.1.5 Preliminary Design. This section and its subsidiary paragraphs requires the

contractor to describe the architecture of the system. (The paragraph is labelled *Preliminary Design*; it would be better to change the label to *System Architecture*.)

Presumably the intent of an SDD is to describe the operational design of a CSCI (rather than the structure of its source code). That is, an SDD is intended to describe how a CSCI is intended to work. Thus this section presumably requires the developer to describe the object structure rather than the CSCI's object description structure.

Yet there should also be a requirement that the developer describe the organization of the object descriptions. In most cases (and whether or not an object-oriented programming language is used) the object descriptions will be organized as a library. The structure of that library deserve adequate documentation.

Paragraph 10.1.6 Detailed design. The subparagraphs within this paragraph require that the design of each operational system component be described. In particular, 10.1.6.1 requires that each higher level operational component be described in terms of the objects within it.

Paragraph 10.1.6.1.2.2 (CSU name) Design. This paragraph requires that the design of a CSU be described.

Since the focus of this paragraph is on objects, it makes sense to argue that this paragraph should be about objects and not about object descriptions. Under that interpretation, this paragraph should require a discussion of the design of the object under discussion. It should also require a discussion of how the object description from which that object is instantiated implements that design.

On the other hand, one may want to require that the object description library be documented. In that case one is not documenting objects for a particular system; one is documenting object descriptions. In some ways this illustrates the problem that underlies many the issues that we are discussing.

- Documentation of an object description is documentation of a source code library element independently of the system within which it is to function.
- Documentation of an object is documentation of an element possibly in the context of the system in which it operates.

In many cases, these two approaches to documentation will yield the same result. But in some cases one wants to document the particular role at that an object plays in a system and the interactions it has with other objects of that particular system. Such documentation is not possible at the object description level since at that level one is documenting source code that may be used in a number of different places in possibly many different systems.

Multiple operations. This paragraph poses a separate but important problem for object-oriented software. There is an unstated implication that each object performs only one operation. In object-oriented software, objects may be able to perform many different operations. The details of each operation should be described in terms of the relevant design properties [(a) through (k)] listed in this paragraph.

Data elements. Portions of this paragraph refer to data elements. In a pure object-oriented system, there are no such things as data elements that are distinct from objects. Data is carried by objects.

Objects may, however, have internal variables to which other objects may be bound. The objects expected to be bound to these local variables should be described.

In describing objects to be bound to local variables it is important to specify the engineering-specific information, e.g., units of measure, limits, etc., specified in paragraph 10.1.7. It is also important to specify for each such object either (a) the type of the object, i.e., object description from which the object is to be derived or (if the system supports polymorphism) (b) the operations that this object is expected to be able to

perform. This information should replace the information on data type and representation required by the paragraph.

Paragraph 10.1.7 CSCI data. This section requires that global data elements be described. As just indicated, in pure object-oriented systems, data elements are not distinct from objects. Therefore this section may not apply. On the other hand, there probably will be global objects that should be described as part of the CSCI architecture.

5. Summary and Recommendations

We have argued that software systems have two structures and that both of them are important. We have found that DOD-STD-2167A requires information about both structures but that it does so in a way that does not recognize that the two structures are distinct. We therefore recommend that recognition of the two structure be included in the upcoming revision to the standard.

Acknowledgments and Disclaimers

Marv Lubofsky originally suggested the need for this investigation, and discussions with him helped to clarify many of the issues. Suellen Eslinger, Peter Homeier, and Larry Jordan also provided helpful comments.

The positions, analyses, and recommendations presented here are strictly those of the author and do not necessarily represent those of any institution with which the author is associated or of any of the individuals who read and commented on drafts of this paper.

References

1. Abbott, Russell J. *Configuration Management, Functional Decomposition, and Object-Oriented Software*. Draft (1991).
2. United States Department of Defense. *DOD-STD-1815A Ada Programming Language*. Department of Defense, January, 1983.
3. United States Department of Defense. *DOD-STD-2167A Defense System Software Development*. Department of Defense, June, 1985.
4. United States Department of Defense. *DI-MCCR-80012A: Software Design Document*. Department of Defense, February, 1988.

Author Information

Russell J. Abbott is a Professor of Computer Science in the Department of Mathematics and Computer Science at California State University, Los Angeles, Ca. 90032. He is interested in various aspects of Computer Science including software and system organization. He can be reached by email at rabbott@neptune.calstate-la.edu.

MATHEMATICS PLACEMENT TESTING: A STUDENT PROJECT

Dr. Dennis S. Martin

Department of Computing Sciences
University of Scranton, Scranton, PA 18510-4664

Abstract --This paper describes a programming project designed for junior year students which emphasizes an object orientation and software engineering concerns. It uses Ada features in a meaningful manner to support these considerations.

keywords: student project, software engineering education

Introduction

One of the problems facing instructors of computer science is the creation of meaningful student projects. The project described here is a class-tested, effective project which requires an interplay between top-down and object-oriented design and which utilizes many Ada features and software engineering considerations. This is a major project which takes over one-half of the semester for students to perform. It is a reasonable project for junior or senior year students. We use this type of project as one of two projects in a Programming Languages course stressing software engineering concerns. The second project is a smaller version of the project in C. A more appropriate usage would be as the project in a software engineering course, possibly using a team approach. Next year, we will introduce such a course and we will modify our approach to the Programming Language course.

Background

The project is derived from an existing software system. Several years ago, we noted that some of our computing sciences students were being placed in Calculus when they needed Pre-Calculus. The use of aptitude test scores and high school records was not working. With the mathematics faculty, we devised a multiple-choice placement examination with a computerized interpretation of scoring. The mathematics faculty used this successfully in all beginning mathematics courses and the University now conducts mathematics placement testing for all incoming freshmen during a two-day pre-orientation in the summer.

A three-part process is involved. First, information about the students is extracted from the University database and this file is incorporated into a testing database. Two mathematics examinations are given on the first day the students are on campus. Students are assigned to an examination based on their choice of major and what mathematics it requires. The tests are optically scanned and the resulting test data file is incorporated into the testing database. Reports are then produced the day of the test and used by advisors to place students into proper mathematics courses.

This process is repeated at five different times during the Summer. Additional reports based on all the students tested are generated in the Fall in various formats for the use of mathematics instructors, advisors, major departments, and administrative personnel.

The Student Project

The project simulates a single test form, a single test administration, and fewer reports but otherwise is faithful to the original system. We use disguised actual student data to test the students' programs.

The project is given to the students within the first two weeks of the semester in a descriptive form such as we have done above. The students have used Modula-2 as the principle language of instruction for four semesters and are learning Ada at this time. Since we use a Language Sensitive Editor, the Ada instruction concentrates on concept rather than syntax.

Design

The students have two weeks to prepare and submit their design in the form of successfully compiled listing files. The design consists of

1. The mainline which is a menu selection for the major procedures. It must be able to be run more than once as the two files are not available at the same time and reports need to be generated on demand. Full system documentation is required.

2. The specifications for the major procedures. Each mainline procedure describes a functionality of the system. A careful English description of each unit is required. These must be fully commented. In each, the access

to the student database must be specified.

3. The bodies of the major procedures with the design filled in. A pseudocode of the mainline of the procedure is the recommended method. A dummy stub allows compilation.

4. The specification of the student database package as an abstract data type. This consists of open and close operations and the access procedures, with parameters, needed for the rest of the program. The testing database is the object that provides logical access to all the information.

The use of separate compilation units is mandatory. The work above involves only a small amount of code and the students cannot develop the entire project to hand in the design. This is intentional as most of the students have never produced a design without using the expected body of code to guide it. We are trying to force design to precede code with this project.

We use two criteria to grade the design. The clarity of the documentation is as important as how well the database provides the access needed for the main units. Adherence to departmental documentation standards is required (the Language Sensitive Editor facilitates this). Students are encouraged to discuss ideas for design with each other. They have trouble with the design, especially the role of the database. Many spend much time developing details of the implementation of the database instead of its specification. The design grade is awarded when the design is returned and is twenty percent of the total grade. After the designs are graded and returned, we review top-down and object-

oriented design for this example and develop a correct design in class and students whose design are unsatisfactory must switch to this new design.

Implementation

The students now start developing the bodies of the various sections. At this point specific Ada constructs are explicitly required. In class, we show Ada code using many of these constructs. Students were free to discuss any problems together but code had to be original with each student. They could ask any Ada questions in class or in my office.

The storage mechanism we use is an instantiation of `Direct_IO`, a direct access file indexed by record numbers. The program must be reasonably efficient as it is run for about 1000 students. Access considerations require direct access by student ID and sequential access alphabetically by name. They must write a generic sort (on an array of key field-record number pairs) which is instantiated twice.

Of course, there are errors in the data. The files downloaded from the University database are clean but are generated prior to the test date. Some students tested were admitted after the file was generated and were not in the file. The test data file has errors caused by the students as they fill out the answer sheet. The student ID number may not be correct and the name may not literally match the student's name in the University database. When a student ID is not found by the database code it must be reported to the mainline procedure that asked the database for the service. In Ada, the natural mechanism is to raise an

exception that will be handled back in the mainline procedure.

The program must test for and recover from all possible data mistakes. The user and the program must "save" as much data as possible by cross-checking or by prompting for extra information from the terminal. What can be done effectively by code and what is best done interactively is a good discussion topic.

Some reports required mean and standard deviation of scores. This required work with float variables and the mathematics library.

We further require that there must be only one procedure which will open all text files whether for reading or for writing and this procedure must recover from an incorrect file name. The purpose of this requirement is to reinforce the corresponding Ada features.

Results

The grading is based on project demonstration and the quality of the code submitted. This project has been assigned twice (with a year's gap in between) to a total of thirty students. Last year's students had actually taken the placement test as incoming students. There are clearly demarcated partial projects that a student can receive credit for. Eight students have done excellent work, producing fully working systems. An additional twelve have done significant work but did not totally finish the project. The most common problem was the inability to correct a wrong student ID or to add a new student. The minimal project that could obtain credit was to read the original file into the database and then print it out in any manner. This

was sufficient to allow a passing grade in the course if the second project and the tests were good. Since the major reason for doing poorly was waiting until the last minute to start the work, the smaller and easier second project was usually done well.

All students find this project to be a major challenge. The best students enjoy the challenge and are proud of their accomplishments. It is not expected that all students will finish this project and that fact is constantly repeated. The students who do significant work but not the whole project are good students and it is essential that they also have a feeling of pride in their accomplishments. Since our classes are small, I can work individually with these students and help and encourage them as they progress. I am proud of what they are accomplishing and I believe that they are also. The students who are having problems and who do not come to see me even when I constantly ask them are the problem. Some know that the grading policy allows them to do minimal work on this project but still pass the course.

We find that the project is an excellent preparation for the individual computer projects that our students must undertake in the following semester as seniors. When asked to anonymously comment on the project, one student made the following points:

1. This project is a very realistic practice of the type of problem a computer programmer will experience in his/her professional life.

2. While I was working on this project I got to appreciate the importance of taking time to find and understand the correct specifications and requirements to

solve the problem in an efficient and reliable way.

3. This project gave me practice to go over some implementation decisions (storing, sorting and searching large amounts of data) that required taking into consideration what I had learned in previous courses.

4. This is a very long project and I got to appreciate the importance of dividing and organizing the job into distinct and approachable parts that can be implemented and tested separately.

This project works well for our students. We feel that you will find a variation customized for your needs similarly useful in your program.

Dr. Dennis S. Martin, Department of Computing Sciences, University of Scranton, Scranton, PA, 18510 martin@jaguar.uofs.edu, is an Associate Professor who teaches a variety of courses and writes papers on effective presentation of topics in computer science.

MAINTAINING TRANSPARENCY OF DATABASE OBJECTS OVER NETWORKS IN ADA APPLICATIONS

Eugen N. Vasilescu
Sabah Salih
John Skinner

Grumman Data Systems
1000 Woodbury Road
Woodbury NY 11797
MS D12-237

Abstract -- Seamless integration of applications in existing MIS environments is difficult because of specific differences in hardware and software characteristics. We investigate the problem of accessing databases and the transparent transfer of complex objects between Ada programs across several hardware platforms utilizing multi-vendor DBMS products. We focus on integrating related data and data manipulation services that often must reside on different platforms, but generally are easily interconnected via off-the-shelf offerings such as NFS (Network File System) and RPC (Remote Procedure Calls). We present a uniform Ada-based methodology that supports these services, and the associated implementation challenges.

Keywords -- Ada, DBMS, Remote Procedure Call (RPC), High-level Schema, Network File System (NFS), Transparent Database Objects, Complex Objects.

1. INTRODUCTION

The establishment of an environment in which several client and server machines may operate is a critical step in permitting communications between different machines. NFS is a de-facto standard when sharing of files is needed in a heterogeneous environment of machines, operating systems and networks. It is the dominant facility used by UNIX and it is widely supported by non-UNIX operating systems such as DOS, OS/2, NetWare. NFS allows a client program to access files on any server on a remote system that supports the NFS protocols and access methods as specified in [1].

Despite the easy establishment of a common file system by NFS, problems can arise when programs try to access and utilize data residing on different server platforms due to differences in internal data representation formats.

Another problem is encountered when data needs to be transferred between databases, possibly maintained by different DBMS products. DBMS vendors often provide utilities for loading and transferring data (eg. Oracle SQL*Loader) [2], but these products can be somewhat complicated to use and may not produce formats that are compatible between products from different vendors.

In a multiple client environment, the problem of data transfer may be further complicated by the need for data communications between application programs running on different client machines. In this case, a transparent method for communicating potentially complex data types is highly desirable.

RPC helps in solving the above problems because it enables communication across a network between programs written in a variety of high-level languages, and can be used to communicate between processes on the same or different machines. The net effect of programming with RPC is that programs are designed to run within a client/server network model [1]. In particular Database, Management Systems operating in a client/server environment can utilize the services provided by RPC to transparently make their services available to processes on different machines.

RPC can be used to address the problem of data transfer between databases that can reside on different platforms. For example, a client application program could use RPC to retrieve data on a server running a DBMS. This data could then be inserted into the client's database or into a database residing on a different server. RPC would automatically make the data conversions necessary for the data transfer. In a similar fashion, RPC could be used to directly transfer potentially complex data types between application programs running on different platforms.

The benefits of the RPC mechanism come with some associated obstacles in its use due chiefly to the tedious manual process of developing the required support code. RPC itself is an extension of XDR (External Data Representation) language, with a bias towards the C Language, and represents an added layer of complexity for the application programmer to contend with. As more complicated data structures are supported, the RPC code becomes exceedingly complex and error prone.

We present an Ada-based methodology that automatically generates DBMS and RPC support code for complex objects. This methodology, first described in [3] for databases, is extended to RPC in a seamless manner. When using this methodology, an application programmer places in a specialized package the types requiring RPC and/or database support. These types appear as parameters of predefined and semantically well-defined high-level calls available to the application program. Every object declaration of the supported types in the application program may use the predefined DBMS and/or RPC calls in a transparent manner.

This methodology is now implemented in a prototype running ORACLE on SUN 4 and INFORMIX on SCO UNIX. The application can run on either platform and either DBMS can be used as a server.

2. METHODOLOGY OVERVIEW

At a conceptual level the methodology requires capabilities for modeling classes of complex objects and associated high-level operations in Ada. It seems (as shown in [4]) that for a range of database models, (hierarchical, network, relational, semantic, object-oriented) what is needed is the capability of describing DAGs (Directed Acyclic Graphs) in Ada. This is accomplished by defining record types with discriminants and the use, under certain restrictions, of access types.

Graph nodes are represented by record types, while graph edges are represented by discriminant values or record components of access type. Careful rules must be observed to avoid the creation of unintended cycles that translate into endless loops at run-time. These rules (outlined in [3]), are obeyed by the PARTS_SCHEMA package listed in the Appendix. Associated high-level primitive for database applications include operations such as INSERT, RETRIEVE, DELETE.

From the user perspective, the methodology requires first that a collection of candidate scalar types be defined in a separate package. These are the types used in the building of desired complex objects. A good source of scalars to be used by the RPC and/or DBMS facilities can be found in data dictionaries or repositories, and their casting into Ada type definitions is usually straightforward. An example of such a package is PARTS_TYPES listed in the Appendix.

Next the collection of supported types is assembled in a separate package. For database applications this collection of types (found in the PARTS_SCHEMA package) describes a schema. This means that any object declared in the application program whose type is listed in PARTS_SCHEMA can be used as an actual parameter in calls invoking RETRIEVE, DELETE, INSERT.

When invoked, RETRIEVE will actually assemble all required components, list of suppliers, etc. consistent with schema description, from the persistent storage under the control of DBMS and place this information in memory under the control of the application program. The RPC support means that the user may transparently access servers on different platforms.

In this case, for instance, the application invoking a RETRIEVE may run on an INTEL 486, while the accessed DBMS might be running on a SUN Sparcstation. The user actually has the option of naming the server ("retrieve this bike from the SUN server") or ignoring server names ("retrieve this bike - I do not know or I am not interested what server has to assemble this information"). Or an application program can retrieve objects from one DBMS and insert them into another.

The RPC support is not tied to DBMS support. One can have several application programs running on different platforms exchanging directly complex objects without any database interaction. As depicted in Figure 1, there are quite a few options in deciding the desired

combination and interconnection of servers and clients.

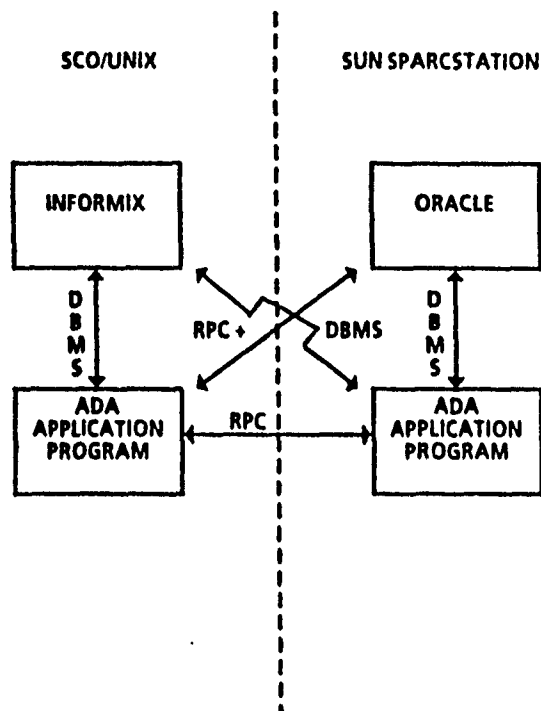


Figure 1

This methodology is heavily biased towards ease of use. The user needs to know a few rules for consistent definition of Ada record and access types making up the complex objects, and have a good understanding of the high-level primitives. The real burden is on the implementor who must generate on-the-fly support code for navigating databases and using RPC. The implementor needs a tool that, at a minimum, is able to perform static semantic analysis of Ada source code. By analyzing the application program and the types making up complex objects, the tool must generate a "virtual interface" capable of mapping complex objects onto simpler ones (required by databases), and also to map the same complex objects onto equivalent RPC ones (an example in the Appendix lists the RPC code generated from the PARTS_SCHEMA package.)

It should be noted that the "virtual interface" generation (see figure 2) is different from the working of a preprocessor in that the application source code is not modified (except for the addition of a few representation clauses), rather the bodies of some "withed" packages are filled out. This can be likened to a "pre-compile time polymorphism" concept, and accounts for the seamlessness of the methodology.

This methodology is independent of any particular DBMS model, and the extension to RPC is done in an orthogonal manner. The prototype implementation uses Relational DBMS, but other models can be targetted as well because of the complexity of the data structures supported. One may choose to have schemas spanning several packages, each package specialized in accessing a particular DBMS or file system.

The methodology offers the application level programmer all the required semantics for writing fully portable, all-Ada code. There is no difference in Ada source code between the application running on SCO/UNIX and the one running on the SUN platform.

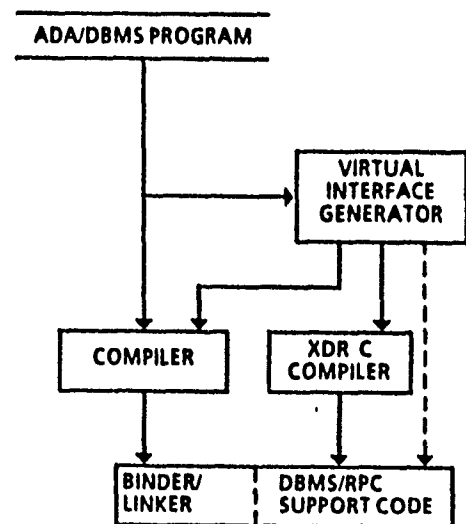


Figure 2

3. PROTOTYPE IMPLEMENTATION ISSUES

The task of transparently transferring complex objects between Ada programs across several hardware platforms utilizing multi-vendor DBMS products encompasses two parts. First, one must be able to successfully move data between platforms transparently to the user. Secondly, access to databases and data manipulation services residing on different platforms must be presented to the user in a uniform fashion that is independent of the specific DBMS's. Our methodology applied to these tasks has resulted in a system that allows for the definition and manipulation of

complex objects in Ada programs utilizing multi-vendor DBMS products that may reside on different machines.

Differences in internal data representation formats result in major problems when binary data is transferred between different machines. This becomes apparent when one understands that the most significant bit for binary integers on INTEL 486 becomes the least significant bit on the Sun SPARCstation. A sequence of bytes storing a certain value on the INTEL 486, will represent a different integer value when copied on the Sun platform.

This is easily demonstrated by using Ada I/O facilities provided by the `DIRECT_IO` package to write a complex Ada record to a file, and then attempting to reconstruct the original record by using `DIRECT_IO.READ` to retrieve the data from the file. When these operations are performed on the same machine, the original record can be successfully read back. However, if the data file is moved to another platform and then read by `DIRECT_IO.READ`, the original record cannot to be reconstructed.

RPC is designed to address this data transfer problem by automatically making the data conversions necessary for the data transfer. However, RPC is typically used to transfer data between C programs on different platforms. Using RPC to implement data communications between Ada programs on different machines presents additional challenges. The goal of providing these services transparently to the user further complicates the issue.

The implementation of the prototype covered the following phases, each one with its specific challenges.

A. Establish communication between Ada programs on different machines and pass simple pieces of data such as integers or strings.

The only available RPC support was for the C language, so the low-level calls were made to C. A problem encountered in this phase was due to the Ada compiler propensity to generate three bytes for integers when the type definition required less than 24 bits for representation. The outside environment had either two or four byte integers. The solution was to use a length clause to allocate 32 bits.

B. Establish a canonical mapping from Ada record type definition to RPC equivalent data structures.

This phase ran pretty much concurrently with the previous one. The issue was to choose a target data structure that closely follows its Ada counterpart (especially when variant records are involved), both for ease in generating automatically RPC support, and for the ability to provide easy visual checks. An example is provided in the Appendix which lists the `PARTS_SCHEMA` package and its RPC translation.

C. Pass complex Ada record structures from an Ada client to an Ada server and back.

When mapping Ada records from an Ada client to an Ada server the record components cross several address spaces, including a thin C layer, and the Ada record layout does not necessarily conform to the record layout of the external environment. Some of the extra gaps between Ada record components were handled with "pragma PACK".

Using a representation clause to control the Ada record layout seemed like another option in this case, but here one has to pay attention to the alignment conventions for record components used by the Ada compiler and the environment. For instance, our Ada compiler started record components in locations divisible by four relative to the start of the record (word boundary alignment), and this might force the introduction of slack bytes when strings whose length is not divisible by four are used in records.

Ada variant record maps naturally on the XDR discriminated union, which in turn gets translated into C unions together with enumeration values that select the proper arm of the union. Again, problems arise here because the Ada compiler chooses the storage size for the enumeration type of the discriminant based on its range of enumeration values.

In `PARTS_SCHEMA` package, `PART_C` record type has two discriminant choices, and the Ada compiler chose 8 bits of storage for the enumeration type. However, the C compiler uses 32 bit integers for its enumeration types. At first, the solution seemed to be to place the discriminant value in the first 8 bits of the Ada record, but allocate a total of 32 bits to match

the size of the C enumeration type. The solution to this problem is to use a length clause for the Ada enumeration type that is used for the discriminant type and force the size to 32 bits. In the Ada variant record, the representation clause now places the discriminant value in the first 32 bits of the record and continues to allocate 32 bits of space for the discriminant component.

D. Mechanize translation of Ada record type into RPC equivalent data structures.

This phase took advantage of the tool developed for databases [3] which already performs the required semantic analysis of the Ada source code and is able to manipulate the graph-like structures for complex objects. Given the availability of the required information in a convenient form, it was a fairly trivial matter to target for translation the RPC language.

E. Integrate RPC and DBMS support.

According to [3], the user supplies subprogram specifications for any necessary data manipulation, and the interface generator will supply the necessary bodies for database navigational code. The DBMS-specific packages that interface to the database exist at a lower level not visible to the user. The same holds for the RPC components, and this allows us to incorporate the RPC support without user intervention.

A significant problem in the integration of DBMS and RPC was caused by the fact that pragma INTERFACE is insensitive to the formal parameter profile.

For instance, the database calls use the same RETRIEVE subprogram name, but each RETRIEVE has a unique parameter profile because different query terms correspond to different formal parameter types. Pragma INTERFACE was required for low-level bundling and unbundling of parameters on the client and the server side and it cannot overload subprogram names. The chosen solution was to embed the required pragma one level deeper in the body of the RETRIEVE subprograms, in effect circumventing the overload resolution by using lower-level scopes.

The resulting prototype successfully supports both RPC and database calls. Before the Ada source code reaches an Ada compiler, our tool will augment the type package (PARTS_TYPES package) with necessary length clauses and the schema package (PARTS_SCHEMA) with representation clauses. The same tool will supply missing Ada code in subprogram bodies for database and RPC support, and additionally, will generate a small layer of RPC and C code. The resulting code is compiled (by Ada, RPC and C compilers) and linked without user intervention.

4. CONCLUSIONS

There are Ada compiler vendors that offer RPC support, but the support offered is for elementary types such as integers, float, strings. They do not offer any automatic support when dealing with the complex objects involving records with discriminants and pointer semantics.

Similarly, some DBMS vendors offer Ada interfaces to their products (usually following an "embedded approach" of placing database language calls in the Ada source code) that are vendor specific and offer limited portability.

In contrast, our methodology offers an all-Ada approach that provides a uniform and transparent user-oriented strategy for dealing with a variety of DBMS and complex objects residing on different platforms.

This methodology is independent of existing commercial offerings, thus enabling portability of user applications. This, in fact, was convincingly demonstrated by our prototype implementation.

While the methodology assumes the use of a proprietary tool, the interfaces between the user code and tool are supposed well understood. This is due to the public nature of the rules for defining complex objects and the general acceptance of the high-level call semantics.

REFERENCES

1. "Network Programming Guide", SUN Microsystems, Mountain View, California, 1990.
2. "Oracle RDBMS Utilities Users Guide", Oracle Corporation, Belmont, California, 1989.

3. Vasilescu, E., "Using Ada for Rapid Prototyping of Database Applications", Proceedings of the Eighth Washington Ada Symposium, pp 40-49.(1991).

4. Zicari, R., "Primitives for Schema Updates in an Object-Oriented Database System: A Proposal", Proceedings of the Object-Oriented Database Task Group Workshop, E. Fong, Ed., NISTIR 4488, January 1991.

APPENDIX

package PARTS_TYPES is

package ADA_SQL is

```

type O_COUNT_TYPE is range -10000..10000;
type PART_CHAR is new CHARACTER;
type PART_INDEX is range 1..10;
type SUPPLIER_CHAR is new CHARACTER;
type SUPPLIER_INDEX is range 1..10;
type CITY_TYPE is (PARIS,LONDON);
type C_TYPE is range 400 .. 500;
type MASS_TYPE is range -1_000_000..1_000_000;
type NO_TYPE is range 0 .. 99999;
type COST_TYPE is range 1 .. 1000;
type PART_NAME_TYPE is array
(PART_INDEX range <>) of PART_CHAR;
type SUPPLIER_NAME_TYPE is array
(SUPPLIER_INDEX range <>) of
SUPPLIER_CHAR;
type HOW_MANY_TYPE is range 1 .. 99;
--
type PART_CLASS IS
(BASE_PART,COMPOSITE_PART);

subtype BASE_PART_CLASS is PART_CLASS
range BASE_PART .. BASE_PART;
subtype COMPOSITE_PART_CLASS is
PART_CLASS range
COMPOSITE_PART .. COMPOSITE_PART;
--
-- These length clauses are added by our tool
-- based on the compiler and environment knowledge
--
for O_COUNT_TYPE'SIZE use 32;
for PART_INDEX'SIZE use 32;
for SUPPLIER_INDEX'SIZE use 32;
for CITY_TYPE'SIZE use 32;
for C_TYPE'SIZE use 32;

```

```

for MASS_TYPE'SIZE use 32;
for NO_TYPE'SIZE use 32;
for COST_TYPE'SIZE use 32;
for HOW_MANY_TYPE'SIZE use 32;
for PART_CLASS'SIZE use 32;

```

end ADA_SQL;

end PARTS_TYPES;

with PARTS_TYPES; use PARTS_TYPES;
package PARTS_SCHEMA is

use PARTS_TYPES.ADA_SQL;

```

type MADE_FROM_C;
type ACCESS_MADE_FROM_C is
access MADE_FROM_C;

```

```

type SUPPLIERS_C;
type ACCESS_SUPPLIERS_C is
access SUPPLIERS_C;

```

--PART_C is your only root

```

type PART_C (PART_KIND: PART_CLASS
:=BASE_PART) is

```

record

```

NO_REF : NO_TYPE;
NAME : PART_NAME_TYPE(1..10);

```

case PART_KIND is

when BASE_PART =>

```

COST : COST_TYPE;
MASS : MASS_TYPE;
SUPPLIED_BY : ACCESS_SUPPLIERS_C;

```

when COMPOSITE_PART =>

```

ASSEMBLY_COST
: COST_TYPE;
MASS_INCREMENT
: MASS_TYPE;
MADE_FROM :
ACCESS_MADE_FROM_C;

```

end case;

end record;

type ACCESS_PART_C is access PART_C;

```

type MADE_FROM_C is
  record

    HOW_MANY      : HOW_MANY_TYPE;
    COMPONENT      : ACCESS_PART_C;
    NEXT_MADE_FROM_C :
      ACCESS_MADE_FROM_C;

  end record;

type SUPPLIERS_C is
  record

    NAME_REF      :
      SUPPLIER_NAME_TYPE(1..10);
    NEXT_SUPPLIER_C :
      ACCESS_SUPPLIERS_C;

  end record;

--
-- These representation clauses are added by our
-- tool based on the
-- compiler and environment knowledge
--

for PART_C use
  record
    PART_KIND at 0 range 0..31;
    NO_REF at 4 range 0..31;
    NAME at 8 range 0..79;
    COST at 20 range 0..31;
    MASS at 24 range 0..31;
    SUPPLIED_BY at 28 range 0..31;
    ASSEMBLY_COST at 20 range 0..31;
    MASS_INCREMENT at 24 range 0..31;
    MADE_FROM at 28 range 0..31;
  end record;

for MADE_FROM_C use
  record
    HOW_MANY at 0 range 0..31;
    COMPONENT at 4 range 0..31;
    NEXT_MADE_FROM_C at 8 range 0..31;
  end record;

for SUPPLIERS_C use
  record
    NAME_REF at 0 range 0..79;
    NEXT_SUPPLIER_C at 12 range 0..31;
  end record;

end PARTS_SCHEMA;

```

```

--
-- This is RPC code is automatically generated from the
-- PARTS_SCHEMA package
--
--
typedef struct made_from_c *access_made_from_c;

struct made_from_c{
  int how_many;
  struct part_c *component;
  access_made_from_c next_made_from_c;
};

--
--
typedef struct suppliers_c *access_suppliers_c;

struct suppliers_c{
  char name_ref[10];
  access_suppliers_c next_supplier_c;
};

enum part_class {
  BASE_PART = 0,
  COMPOSITE_PART = 1,
  END = 2};

struct base_part{
  int no_ref;
  char name[10];
  int cost;
  int mass;
  access_suppliers_c supplied_by;
};

struct composite_part{
  int no_ref;
  char name[10];
  int assembly_cost;
  int mass_increment;
  access_made_from_c made_from;
};

union part_c switch
  (enum part_class part_class) {

  case BASE_PART:
    struct base_part base_part;

  case COMPOSITE_PART:
    struct composite_part composite_part;

  default:

```

```

void;
};

typedef struct part_c *access_part_c;

struct part_c_base_part_join {
    struct part_c_join *part_c_join;
    struct part_c part_c_elem;
};

struct part_c_composite_part_join {
    struct part_c_join *part_c_join;
    struct part_c part_c_elem;
};

union part_c_join switch
(enum part_class part_class) {

case BASE_PART:
    struct part_c_base_part_join
    part_c_base_part_join;

case COMPOSITE_PART:
    struct part_c_composite_part_join
    part_c_composite_part_join;
default:
    void;
};

typedef struct part_c_join *access_part_c_join;

program PARTS_PROG {
    version PARTS_VERS {

void
INSERT_1(part_c) = 1; /*subprogram number = 1 */
part_c_join
RETRIEVE_1(void) = 2; /*subprogram number = 2
*/
void
DELETE_1(void) = 3; /*subprogram number = 3 */
    } = 1; /*version number = 1 */
    } = 0x31234567; /*program number =
0x31234567; */

--
-- application program
--
with PARTS_DDL;
with DATABASE, PARTS_TYPES,
    PARTS_SCHEMA;
use DATABASE, PARTS_TYPES, PARTS_SCHEMA;
with PARTS_JOIN_TYPES;
use PARTS_JOIN_TYPES;
with TEXT_IO; use TEXT_IO;

```

```

with ADA_SQL_SUPPORT; use
    ADA_SQL_SUPPORT;
with RETRIEVE_PKG; use RETRIEVE_PKG;
with PARTS_VARIABLES; use PARTS_VARIABLES;
with INSERT_PKG; use INSERT_PKG;
with PARTS_UTILITIES;
with SERVER_PACK; use SERVER_PACK;

```

procedure PARTS is

use PARTS_TYPES.ADA_SQL;

SERVERNAME : SERVERS := sol;

THE_RESULT_LIST : ACCESS_PART_C_JOIN;

THE_PART : PART_C;

THE_PART2 : PART_C(COMPOSITE_PART);

begin

```

PARTS_UTILITIES.CONSTRUCT_BIKE;
PUT_LINE("INSERT THE BIKE");
INSERT(PARTS_UTILITIES.PC3,sol);
PUT_LINE ("RETRIEVE (THE PART," &
" R_A_1(THE PART.NAME = BIKE)); (BIKE)");
THE_RESULT_LIST :=
    RETRIEVE ( THE_PART ,
        R_A_1(THE_PART.NAME =
            "bike "),sol);
-- CHECK WHAT WE GET BACK

```

```

while THE_RESULT_LIST /= null
loop
    PARTS_UTILITIES.DISPLAY
        (THE_RESULT_LIST.PART_C_ELEM);
    THE_RESULT_LIST :=
        THE_RESULT_LIST.PART_C_JOIN;
end loop;

```

DELETE(THE_PART,sol);

end PARTS;

Biographical Sketches

Eugen Vasilescu holds a Ph.D. in Game Theory from the University of Illinois (1975) and a MS in Computer Science from the University of Bucharest (1969). He is the Manager of Ada Lab at Grumman Data Systems where he initiated and currently directs R&D work on

Ada interfaces to COTS and DBMS. He published and consulted widely in the area of Ada use in IS. He is the author of the first textbook to focus on Ada use in IS.

Sabah Sahih holds a MS in Computer Science from North Dakota State University (1983). He is currently a Senior Programmer Analyst in the Ada lab of Grumman Data Systems working on design and implementation of Ada interfaces to DBMS.

John Skinner holds a MS in Computer Science from Rochester Institute of Technology (1989). He is currently a Programmer Analyst in the Ada lab of Grumman Data Systems working on design and implementation of Ada interfaces to DBMS.

SYNTHESIS OF DESIGN METHODOLOGIES FOR Ada

Hilary J. Allers
TRW, Inc.
One Federal Systems Park Drive
Fairfax, VA 22033

Major Charles R. Petrie
U. S. Army
PM AWIS
Fort Belvoir, VA 22060

Abstract

This paper describes work performed on the Army WWMCCS Information System (AWIS) program in integrating Functional Decomposition, Object-Oriented Design and Model-Based Design. The resulting methodology allows engineers to define two abstractions for the same set of software modules: one which describes the implementation of functions, and one which describes the implementation of object classes. These two views of a single system are documented in a tailored version of the DOD-STD-2167A Software Design Document.

1. Introduction

The purpose of a software design methodology is to provide guidance to engineers in making the transition from an abstract representation of requirements to a software structure which will support the project's software goals. Such goals may include the performance characteristics of the software, quality attributes such as maintainability and portability, and managerial concerns such as the ease with which the software

can be formally qualified. Design methodologies can provide procedures, tools, heuristics and examples which purportedly increase the likelihood that the resulting software design will support the software goals.

Functional Decomposition was one of the first formalized software design methodologies [13]. As the discipline of Software Engineering has matured, a number of later software design methodologies have emerged which claim to provide better support to a larger number of standard software goals. These methodologies include variations of Functional Decomposition, variations of Object-Oriented Design [5], and a relatively new methodology, Model-Based Design [6].

Functional Decomposition, Object-Oriented Design (OOD) and Model-Based Design are typically thought of as mutually-exclusive alternatives which could not be undertaken on the same software configuration item at the same point in its life cycle. Several "linkage" methodologies have arisen from this belief which attempt to provide (most

commonly) a transition from Functionally-Decomposed Requirements Analysis to Object-Oriented Design. More recently, attempts have been made to extend OOD into other life cycle phases, such as Object-Oriented Requirements Analysis (OORA).

This paper does not attempt to analyze the strengths and weaknesses of either transition methodologies or OORA. Instead, we will describe a software design approach for Ada which synthesizes Functional Decomposition, Object-Oriented Design and Model-Based Design in order to simultaneously derive the unique benefits of each methodology. Our methodology has been applied to the design of multiple CSCIs within the Army WWMCCS Information System (AWIS) program for PM Strategic Army Command and Control Software (SACCS). This paper will begin by presenting an overview of each design methodology including the benefits and deficiencies associated with using each methodology in isolation. We will then describe our synthesized approach using a design example from the domain of Management Information Systems (MIS). The paper will conclude with a brief discussion of issues related to mapping our approach to DOD-STD-2167A.

2. Comparison of Design Methodologies

2.1 Functional Decomposition

Functional Decomposition was originally used to describe the automation of existing manual processes [13]. Process decomposition ("data-flow")

diagrams showing the increasingly detailed breakdown of higher level tasks into subtasks were used to formalize this description. The idiom of a hierarchical decomposition of processing was particularly natural to users already performing the process to be automated ("First I locate the Accounts Receivable file for the customer, then I look at the last invoice to determine the amount due...", etc.). However, the prescribed software model for the implementation of hierarchical processes was nested subprograms, which did little to address software engineering concerns such as information hiding and modularity.

Nevertheless, the concept of describing a system in terms of the hierarchical functions supported is still useful from both an end-user and a functional test perspective. The description of the software control and data flow which takes place to complete a single high-level function allows users to determine that all required processing has been implemented, and allows testers to define test cases which validate the interfaces and paths within the system. It would be desirable to describe the functional decomposition of the software without being required to organize the software along purely functional lines.

2.2 Object-Oriented Design

Object-Oriented Design attempts to organize both the data and the processing of a system into relatively independent modules which represent objects in the real-world problem space [4].

This design methodology encourages engineers to distinguish between the interface (processing and data accessible to other modules) and the private (non-accessible) portions of a module. As such, OOD provides better support for information-hiding than its precursors.

We have found that Object-Oriented Design generally improves modularity and decreases component coupling. However, OOD has not provided a universally successful, "cookbook" style implementation guide as was present with Functional Decomposition. The process of determining the level of abstraction in the problem space at which objects will be modeled, the identification of objects, the determination of object hierarchies, and the identification of the set of operations for each object is a subjective and iterative process. Furthermore, in cases where requirements definition has been accomplished using a Functional Decomposition approach, the allocation of requirements to OOD modules may be indirect, confusing, and difficult to test. For example, the fact that functions may be distributed across operations on multiple objects means that many requirements will be partially satisfied by all (or most) of the modules in the system.

2.3 Model-Based Design

Model-Based Design was developed at the Software Engineering Institute (SEI) as part of a project in Domain-Specific Software Architectures (DSSAs) [10]. Model-Based Design attempts to identify "recurring

problems" or paradigms in a system and produce a model solution that will be applicable to all instances of the paradigm. For these reasons, we have found Model-Based Design to be particularly conducive to Software Reuse.

Although Model-Based Design is, like OOD, dependent upon subjective valuations of elegant construction, it has the advantage of being inherently an iterative approach. As such, Model-Based Design allows for the evaluation and refinement of a design prior to its large-scale implementation. Because Model-Based Design is focused on achieving reuse, the resulting software modules often resemble software "tools" rather than either real-world objects or hierarchical processes.

2.4 Summary of Comparisons

The advantages of each design methodology can be summarized as follows: Functional Decomposition provides a natural medium for describing the system to non-Software Engineers, OOD enhances modularity and portability, and Model-Based Design enhances reuse. We have created a synthesized approach to software design with Ada which allows us to benefit from the strengths of each of these separate methodologies.

3. Synthesis Approach

Our design approach includes both a methodology for producing the software design, and a means of expressing the design in documentation. Fundamental to our methodology is the concept that design entities can be

represented as abstract (non-executing) collections of software modules. For instance, a Computer Software Component (CSC) is an abstract collection of Ada compilation units - rather than being an actual functioning portion of the system in and of itself. This interpretation of a CSC is consistent with ACM and Government direction for mapping DOD-STD-2167A to Ada [1,8]. We represent such abstract collections using "placeholder" Ada packages. These packages contain no declarations or executable statements. They are used to hold design information in the form of structured comments, including decomposition information, requirements allocation, etc. An example of a placeholder Ada package for a CSC is given in figure 3-1. On AWIS, these placeholder packages are

additionally used by the Rational Design Facility to generate the Software Design Document (SDD). Because design entities are abstract collections of objects, we can simultaneously create multiple abstractions which partition the system in different ways. For instance, we can create one partition which represents the class families within a system in an OOD sense, and another partition which represent functional threads within a system in a Functional Decomposition sense. This two-dimensional view of a single system is depicted in figure 3-2. Our methodology relies on maintaining two such abstractions simultaneously throughout the design process: one abstraction to represent the functions implemented in the system, and one abstraction to represent the

```
--| @COMPONENT_KIND
--| CSC
--|
--| @ORIGIN
--| Internal Development
--|
--| @AUTHORS
--| T. Roth
--|
--| @REVISION_HISTORY
--| [Date] [Name] [SCR#] [Reason]
--|
--| @REUSABILITY
--| (Mobilization, Database)
--|
--| @DESCRIPTION
--| The Database CSC provides all database interface processing
--| necessary for the Mob/ODEE CSCI.
--|
--| @PURPOSE
--| The Database CSC contains all application processing
--| and data declarations which interact with the database through
--| Data Manager and DBIF Support Software. The Database CSC is
--| divided into two sublevel CSCs: Viewdefs, which contains all
--| Viewdef packages for the threads; and Viewdef Maps, which
--| contain all Viewdef Map packages for the threads.
--|
--| @DECOMPOSITION
--| (Viewdefs, Viewdef_Maps)
--|
package Database is
end Database;
```

figure 3-1 CSC Placeholder Package

object classes and/or models in the system.

To aid in explaining our methodology, we present an example taken from the domain of

Management Information Systems. Our example is a system for determining staffing requirements for a school system, and for tracking staffing levels against these requirements over time.

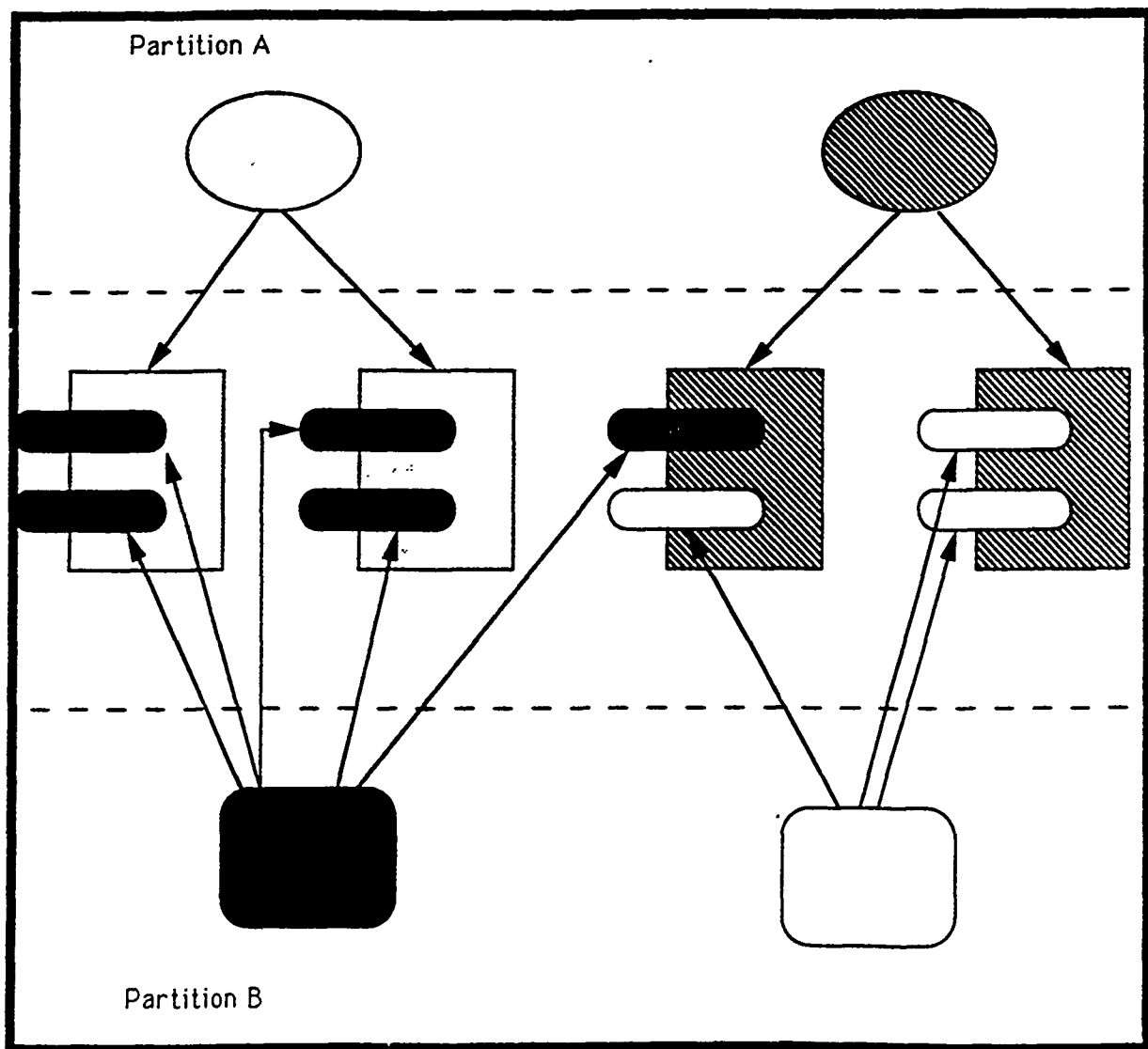


Figure 3-2. Different Partitions of the same Ada Modules

Figure 3-3 presents an abbreviated description of the (Functionally-Decomposed) requirements for this system. Our methodology begins by creating a software abstraction to represent the software requirements. We call this abstraction the "dynamic" (as opposed to "static") structure of the software design. The dynamic structure is represented by placeholder Ada packages, similar to CSCs, which we name Computer Software Functional Threads (CSFTs). A CSFT is simply a software representation of a single function. It contains commentary information about the function, and may be decomposed into the lower-level software elements which are sub-functions.

A low-level CSFT will ultimately be composed of all Ada elements which are executed to complete a given function. These Ada elements which make up a CSFT may be library units, or sub-elements within a library unit, including subprograms, tasks, or single task entries. However, when the CSFT is first created, the underlying software structure which will implement the function has not been determined. Initially, the lowest level CSFTs have no decomposition; They include only design commentary on the purpose of the Functional Thread, and the requirements which are allocated to the thread. This design information will later be augmented with descriptions of control flow,

1. Determine Staffing Requirements

- 1.1 Accept User Input on expected number of students by grade by year for the next 10 years
- 1.2 Accept User Input on expected curriculum area registrations by student by year for the next 10 years
- 1.3 Compute Staffing Requirements by curriculum area by year for the next 10 years

2. Track Staffing Levels to Requirements

- 2.1 Accept User Input of teacher attrition, new hires
- 2.2 Display Summary Report showing shortfalls, over-staffing by curriculum area by year for the next 10 years

figure 3-3 Functional Requirements

data flow and algorithms implemented by the components of the CSFT. Low-level CSFTs can be combined to form more complex functions, or higher-level CSFTs. Figure 3-4 shows an initial CSFT structure for our example School Staffing MIS problem. The CSFTs provide input to a Model-Based analysis as the next step in our design process. CSFT descriptions are analyzed to identify recurring problems in the system. Alternative

solutions to these recurring problems are prototyped to provide performance data and to test design assumptions. The result of the modeling phase is a graphical representation of a standard solution (with possible variations) to the recurring problems. This graphical representation has a corresponding structure in prototype Ada components. The prototype components are frequently re-engineered into

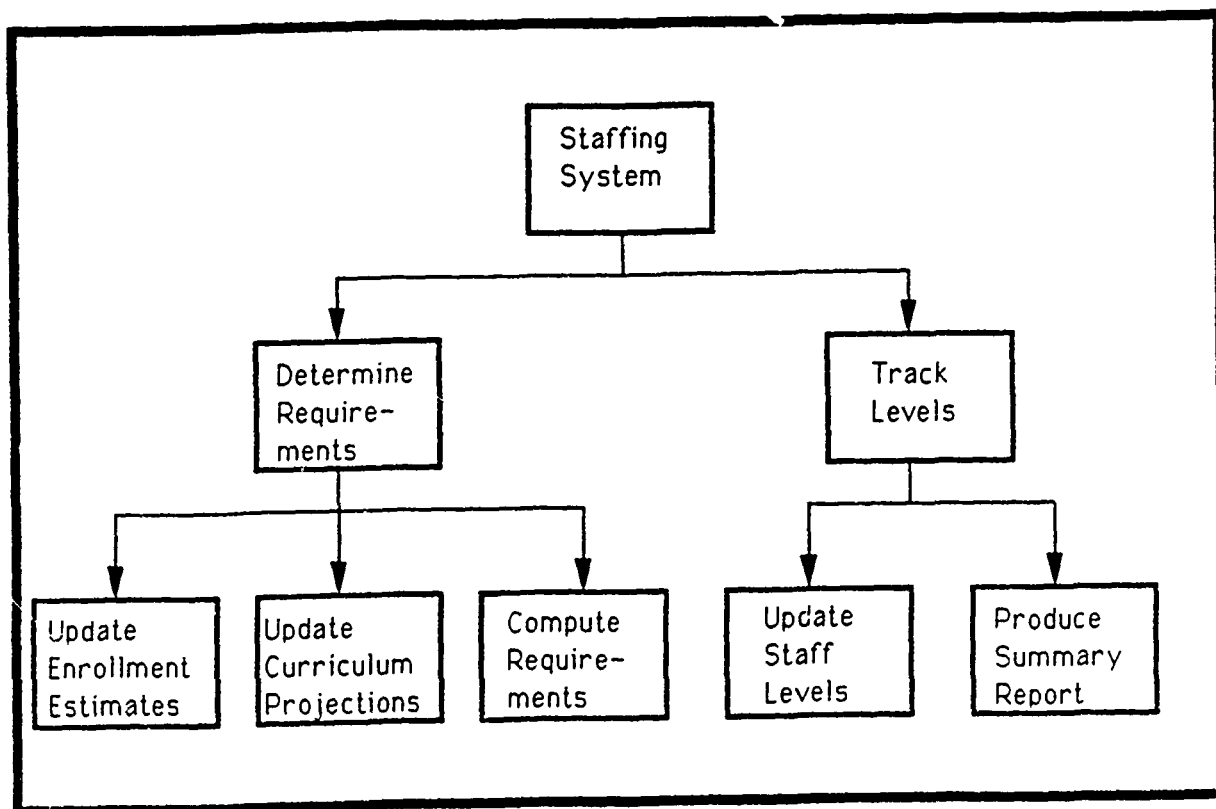


figure 3-4: Initial CSFT Hierarchy

generic units, templates or code generators to enhance the reusability of the model solution. The model solution is depicted using a Model Diagram [10], as illustrated in Figure 3-5. The components shown in Figure 3-5 represent actual MIS components developed as part of the AWIS program to implement a Command and Control MIS system. They represent reusable tools to build applications which accept a user query, retrieve the requested records from a

database, display the results to the user, and (optionally) allow the user to update these results. This functionality can be used to build all of the user update and report functions for our school staffing system. The component building blocks of the model solution form the top-level Computer Software Components (CSCs) of the static structure, as shown in figure 3-6.

The decomposition of each CSC is next defined using a traditional

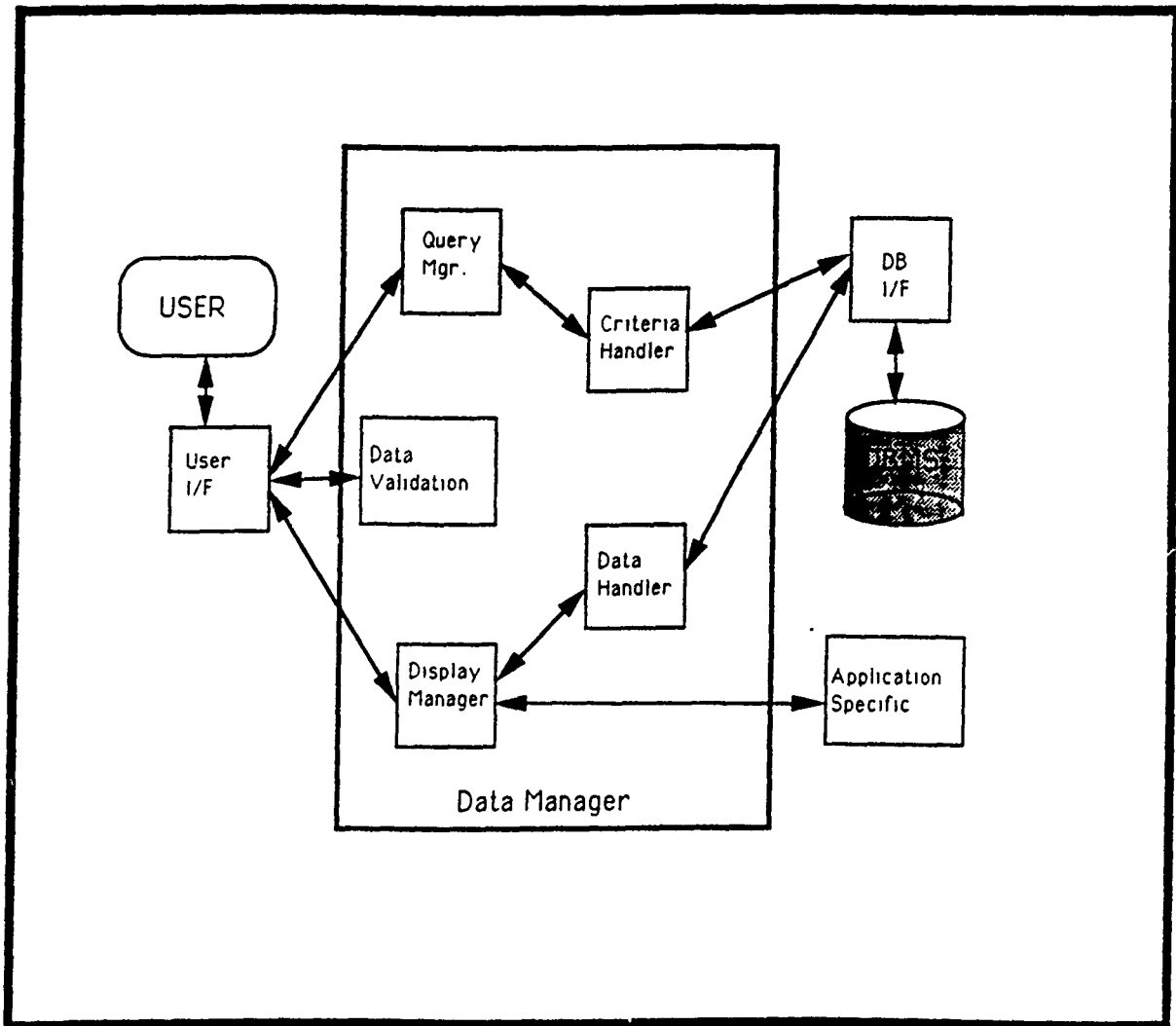


figure 3-5 Model Diagram for MIS Components

OOD approach. During this phase, lower-level CSCs and CSUs which will implement all threads in the system are identified. (AWIS uses a mapping of DOD-STD-2167A to Ada which defines CSUs as Ada library units.) These lower-level CSCs and CSUs implement the object classes and object instances of the system. For instance, our Data Validation

CSC may be decomposed into objects representing real-world entities such as Students, Teachers, and Courses. The information on the correct syntax for Display and Database representations of attributes of these objects (Student Name, Course Title, etc.) are encapsulated within each object package. Operations on these

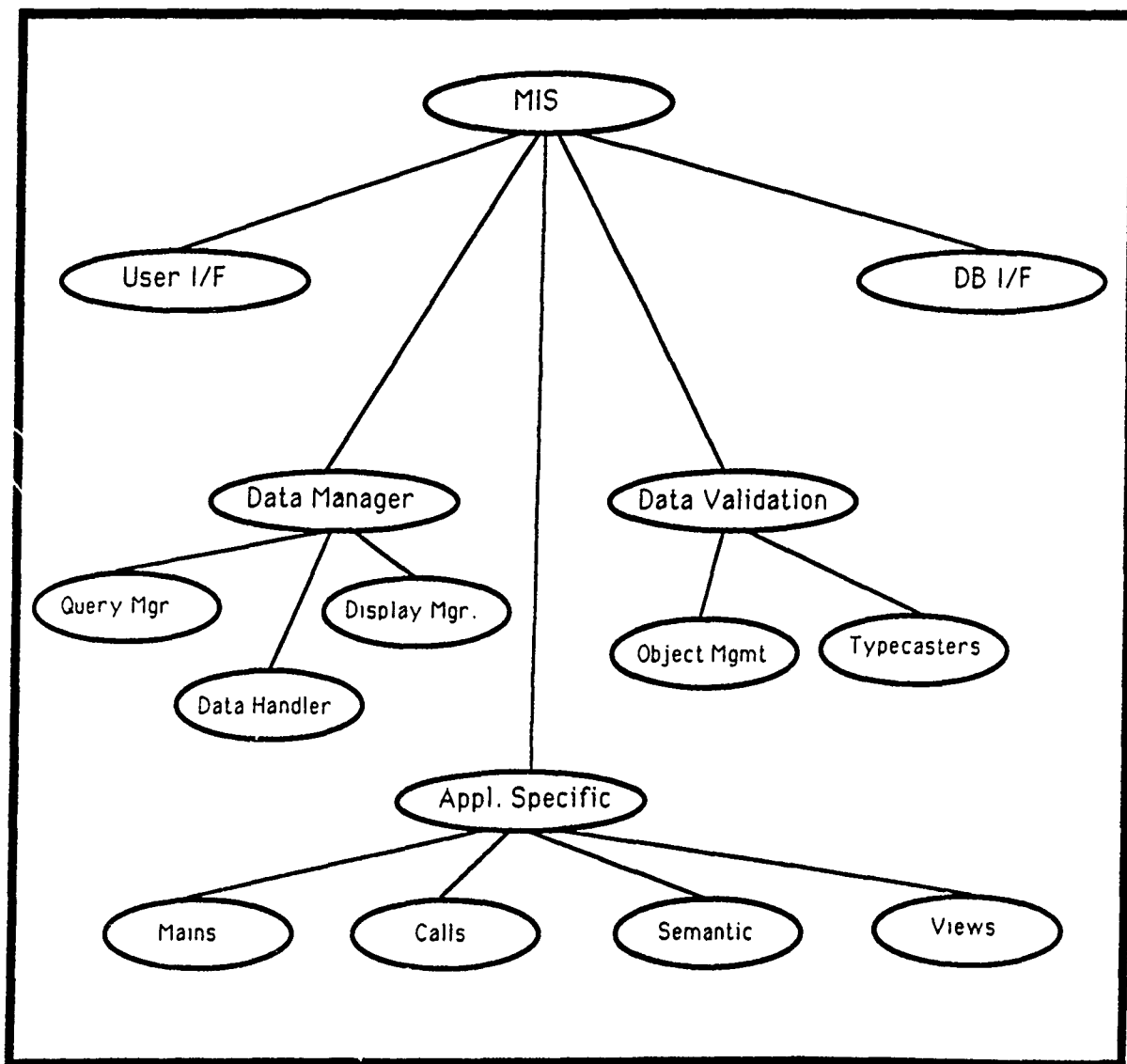


Figure 3-6 MIS Static Structure

objects are provided to validate the syntax of user- or database-provided attribute fields. As the CSUs are built, the components of each CSU which contribute to the execution of each CSFT are identified. All CSFT placeholder packages can then be completed, including decomposition, and descriptions of control flow, data flow, etc. The description of CSFTs are enhanced using Thread diagrams, such as the one shown in figure 3-7. The actual software design is therefore based primarily on Model-Based Design, with an OOD view toward completing each

model. We use our abstract CSFT packages to provide the mapping of system functions in our non-Functionally-Decomposed design. Because both CSCs and CSFTs contain design commentary, we can document design information in the abstract entity where it is most appropriate. For instance, Control Flow, Data Flow, and Performance requirements are most meaningful in the context of a Functional Thread. This information is therefore included in CSFT descriptions. However, information such as Error-Handling rules, Reusability, and Component Interfaces are more

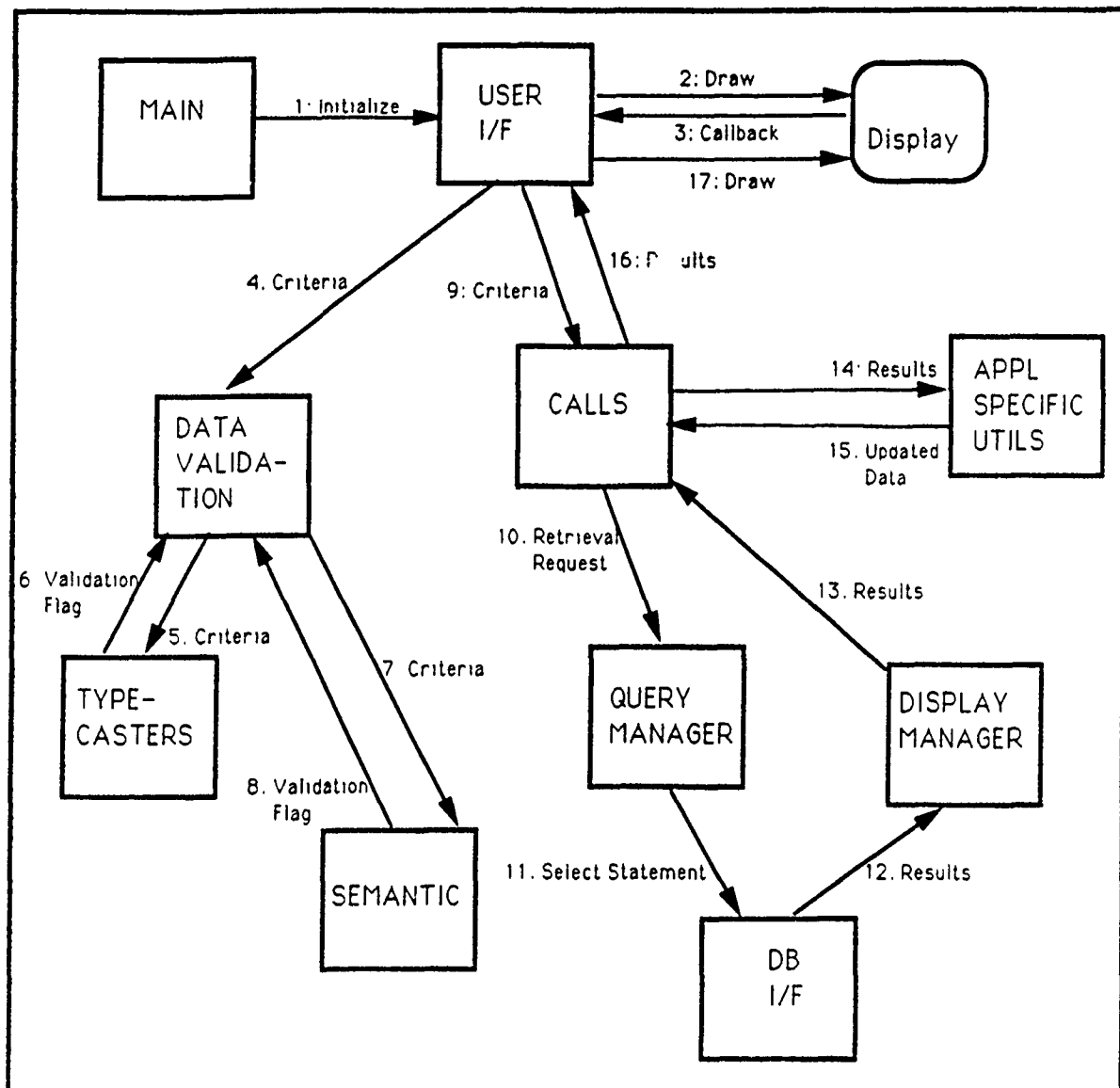


figure 3-7 CSFT Processing Diagram

appropriately discussed in the context of the static structure, or CSCs. Requirements may be allocated to either CSFTs, CSCs or CSUs. In the case of Functional Requirements, these are generally mapped one-to-one with a single CSFT. Likewise, integration test cases are typically documented at the CSFT level.

The SDD is produced incrementally during the entire design process to reflect thread information, architectural information through models, and model instantiation as objects and classes. The SDD is composed of three principal sections: Architecture, Design Overview, and Detailed Design. We have tailored the content of the Architecture section to contain descriptions and diagrams of all models established for the system and a listing of all CSFTs and CSCs which compose the system. We have also tailored the Design Overview section to include both a Dynamic Design subsection which describes each CSFT, and a Static Design subsection which describes each CSC. Inter-CSC interfaces, CSFT hierarchies, and the mapping between CSCs and CSFTs are also included in the Design Overview. The Detailed Design section of the SDD contains descriptions of all CSUs, and is basically unchanged from the 2167A DID.

4. 2167A Implications

Although our methodology represents a substantial tailoring of DOD-STD-2167A, we believe that it actually improves compliance to the spirit of the standard over other mapping strategies described in the

literature [1]. Previous mapping descriptions have struggled with the dilemma of merging a document framework based on Functional Decomposition with an Object-Oriented programming style. This dilemma is derived from language in the standard requiring, among others, that CSCs "execute", have control flow, and implement functions. Our methodology creates abstract CSFTs, whose sub-elements execute and whose descriptions can logically contain all of the function-related documentation required by DOD-STD-2167A. However, since CSFTs are only one partition of the software, the foundation software modules may be organized to support OOD and Model-Based Design.

5. Summary

We have found that the synthesis of Functional Decomposition, OOD, and Model-Based Design provides a structure which allows designers to concentrate on different aspects of the design with a unique set of goals at each phase. Furthermore, by providing the multiple viewpoints of each methodology within a single Design Document, we have increased the amount of useful information about the system in the document, and organized that information so that readers in differing roles (coders, testers, evaluators, maintainers) can readily identify those sections most applicable to their own needs. The size of the document is not substantially increased, as the changes apply only to section 3, a design overview section. We believe that each methodology functions as a tool with a specific purpose, role, and benefit, and that each

methodology contributes to both the preparation and the communication of the software design.

References

1. Association for Computing Machinery (ACM) Special Interest Group on Ada (SIGAda) Software Development Standards and Ada Working Group (SDSAWG), Implementing the DOD-STD-2167A Software Organization Structure in Ada, ACM, New York, NY, August, 1990.
2. Anderson, J. A., Sheffler, J. D., Ward, E. S., "Manageable Object-Oriented Development: Abstraction, Decomposition, and Modeling", Proceedings, Tri-Ada '91, ACM, New York, NY 1991.
3. Bailin, S. C., Bewtra, M., Moore, J. M., "Combining Object-Oriented and Functional Paradigms in a Design Methodology for Ada", Proceedings, Tri-Ada '90, ACM, New York, NY, Dec. 1990.
4. Booch, G. Object-Oriented Design with Applications, Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.
5. Booch, G. Software Engineering with Ada, Benjamin/Cummings, Menlo Park, CA 1987.
6. D'Ippolito, R., "Using Models in Software Engineering", Proceedings, Tri-Ada '89, ACM, New York, NY, October, 1989.
7. DOD-STD-2167A, Defense System Software Development, U. S. Department of Defense, 29 February 1988.
8. MIL-HDBK-287, A Tailoring Guide for DOD-STD-2167A, Defense System Software Development, U. S. Department of Defense, 11 August 1989.
9. Mills, H. D., Linger R. C., Hevner, A. R., Principles of Information Systems Analysis and Design, Academic Press, Inc., Orlando, FL, 1986.
10. Plinta, C. and Lee, K., "A Model Solution for the C3I Domain", Proceedings, Tri-Ada '89, The Association for Computing Machinery, Inc. New York, NY, October 1989.
11. Prieto-Diaz, R., "Domain Analysis: An Introduction", Software Engineering Notes, Vol. 15, No. 2, April 1990.
12. Drake, R., Ett, W., "Reuse: The Two Concurrent Life Cycles Paradigm", Proceedings, Tri-Ada '90.
13. Yourdan, E. and Constantine, L., Structured Design: Fundamentals of a Discipline of Computer Program and System Design, Yourdan Press, New York, NY 1978.

Authors

Hilary J. Allers is a Senior Member of the Technical Staff with TRW, Inc. in Fairfax, VA. Ms. Allers received a Bachelor of Science degree in Computer Science from the University of Maryland in 1985. She is currently pursuing a Master's degree in Computer Science from the Johns Hopkins University. Ms. Allers is the Principal Investigator on a TRW Internal Research and Development Project,

and is an Ada software developer on the AWIS project.

Major Charles R. Petrie, U. S. Army Acquisition Corps, has been a Software Product Line Manager for PM AWIS since 1988. Major Petrie received a Bachelor of Science degree in Engineering from the United States Military Academy, West Point, NY in 1979. He received a Master's degree in Electrical Engineering with a concentration in Computer Engineering from Pennsylvania State University in 1987. He is also a graduate of the Telecommunications Officers Course given by the Air Force Institute of Technology, Wright Patterson Air Force Base.

An Ada Experiment on the Intel Hypercube

Ronald J. Leach

Don M. Coleman

Lu-Ping Tan

Department of Systems & Computer Science
School of Engineering
Howard University
Washington, DC 20059

ABSTRACT

This research is concerned with *N-version programming* in a parallel computing environment. The Ada language is especially well suited to be used in *N-version programming* since it allows the clean modeling of independent concurrently running processes by means of tasks, the communication and synchronization between tasks by means of the rendezvous mechanism, and a smooth method of exception detection and handling.

In this note, we describe the results of porting our research to the Intel hypercube. Efficiency and fault-tolerance of the *N-version programming* system on the Intel hypercube are discussed. Suitability of extensions of the Ada language for a distributed memory parallel environment is also discussed. We also describe some of the problems that are to be expected when porting the *N-version* technique to shared memory parallel computers such as the Alliant.

1. INTRODUCTION

The concept of *N-version programming* was developed by A. Avizienis as a method of increasing the fault-tolerance of software by increasing the level of redundancy. The basic principle of *N-version programming* is that independently created versions of an algorithm are run and the results produced by the various versions are to be compared by an independent

process called the voter process. In practice, the independent versions of the algorithm will communicate the results of certain intermediate computations to the voter process and the voter process will determine the "correct state" of the system by using a majority vote. Therefore a statement that the program is in a "correct state" is actually a statement that the program is in a "consistent state", which simply means that a consensus has been obtained by the voter. The most accessible reference for *N-version programming* is [1], although the basic papers, usually by Avizienis, are much older.

Another approach to fault tolerance is that of Randell [6] in which the program is rolled back to a previous state of execution at which the program is assumed to have been correct. These methods are different from the Ada methods for handling and detecting exceptions ([5], [8]). For a discussion of the differences between these methods for improving fault-tolerance and the ramifications of incorporating them into Ada, see [4].

In a previous paper [2], we described the results of an experiment in *N-version programming* that was performed on two sequential computers: an AT&T 3B2 and a SUN 3/60. Both of these computers implement tasking by allowing interleaving of the cpu cycles; this is not true concurrent execution. This is not quite an ideal environment since each separate version of an algorithm does not have complete access to a cpu. It is clearly better to have an truly concurrent environment in which the indi-

vidual versions are separately running, with each version having its own access to a cpu dedicated to running that version without sharing any cpu cycles. Such an environment is available on the Intel hypercube.

In this note, we consider the porting of the experiment to the Intel iPSC/2 parallel computer. This computer uses the Intel 80386 processor in each of several identical nodes. We will use the terms "node" and "processor" interchangeably. The nodes have equal amounts of local memory; there is no shared memory.

In our system, there are eight nodes, each of which contains four megabytes of memory. The individual nodes run an operating system called *NX/2*, which is similar to a stripped-down version of UNIX. The *NX/2* operating system requires approximately one half megabyte of memory on each node. The rest of the memory on each node is separated into a region for user's programs and data and for the operating system to have message buffers. On our system, each node has 634 message buffers that are used to hold system messages and application messages shorter than 100 bytes. The larger messages must share the remaining physical memory on the nodes.

The Intel hypercube is a distributed memory parallel computer system; this means that each node has its own local memory and that processes running on different nodes can only communicate via messages. The message passing software is part of the operating system for the hypercube and can be accessed from any language by means of special system calls (provided that the language has the right bindings to the operating system).

The only form of communication allowed in this system is the passing of messages from a process running on one node to another process running on the same or on a different node. Messages can be of different sizes and of different types. The type of a message is user-defined and is used as a way of indicating the sender of the message. The host computer, which is the front end of the hypercube, runs the UNIX operating system. Communication from nodes to other nodes and between nodes and the host is handled by the *NX/2* operating system. This is in contrast to the AT&T 3B2 and SUN environments in which all communication between Ada tasks is done using the Ada rendezvous mechanism.

A typical syntax of an Ada language interface to an *NX/2* command to send a message looks like

```
csend(NODE0_TYPE, msg'address, msg'size / 8,  
0, NODE0_PID);
```

(this sends a message named "msg" to a process that is running on node 0 and whose process id is NODE0_PID).

The typical syntax of an *NX/2* command to receive a message is

```
crecv(HOST_TYPE, msg1'address, msg1'size/8);
```

(this allows the host to receive a message whose name is "msg1").

The designs of the system calls are based on facilities available in UNIX and have a C-like syntax. See [7] for more information about the syntax of UNIX messages.

In spite of the name "hypercube", the individual nodes of the Intel iPSC/2 hypercube are logically connected by a mesh topology and can communicate with any other node (not just the nearest neighbor) and with the front end computer called the host.

The Ada implementation on the Intel hypercube is a hybrid system. Ada tasks can be run on the host computer and on any node. Multiple tasks can be run on the same node or host and tasks running on the same cpu can communicate with one another using the rendezvous mechanism. Tasks running on different processors can communicate using the UNIX message passing system call or its *NX/2* analog.

The hybrid nature of the Ada implementation on the Intel hypercube causes some unusual problems and provides several opportunities for research. Natural questions concern the efficiency of *N-version programming*, the fault-tolerance of the message passing and Ada rendezvous communications, ease of programming, scalability, etc.

2. Efficiency of Parallel N-version Programming

There are several questions to ask concerning the efficiency of *N-version programming*

in this parallel environment. As is typical in parallel computation, we define the *efficiency* of parallel computation as the ratio

$$\text{Efficiency} = T(1)/T(N)$$

where $T(1)$ is the time for the performance of the computation using the best sequential algorithm on one processor and $T(N)$ is the time for the performance of the same computation using N processors. The *speedup* is defined as the ratio

$$\text{Speedup} = \text{Efficiency} * N$$

or

$$\text{Speedup} = N * \frac{T(1)}{T(N)}$$

The *efficiency* of the parallel computation must be compared with the *overhead* of *N-version programming*. The overhead of *N-version programming* in any environment, parallel or not, includes the time to perform the computations of the individual versions of the software, to communicate the results of the intermediate computations to the *voter*, to have the *voter* make the decision about the correct state of the computation, and for the *voter* to take appropriate action if one of the versions consistently produces results that the *voter* deems to be correct in the sense that the results of the versions are consistent. The *voter* also must take action if one of the versions aborts or hangs up.

In a sequential computing environment, the time for the execution of the various versions can be quite long since the versions, and the *voter*, must share the cpu with one another. There is also the additional overhead of many context switches which must occur when a process becomes active (gets the cpu) and another is suspended. The cost of a context switch is quite high in a UNIX environment since the process being switched out has to have its intermediate state stored, including the contents of the system stack and the program counter. Since most UNIX implementations of Ada include all tasks within a single UNIX process and the switching of tasks does not require a

context switch, the relative overhead of Ada tasking is smaller than that of simulating concurrent programming in a language such as C in which UNIX processes must have more context switches. These efficiency issues for Ada and C implementations in a sequential environment were discussed in [2] and [3].

In a parallel computing environment, the time needed for the execution of the various versions can be shortened since the versions are presumed to be working in parallel with one another and with the *voter*. The computing times for the versions are not added together as in the case of a sequential machine; instead, they are bounded by the longest time for a version to provide a result. Note that the computing time for the *voter* to determine the correct state of the system can also be ignored since the *voter* can proceed while the versions are performing their next computations. The times for context switches are also not needed.

It is clear that, because the versions can execute simultaneously, some of the time needed for *N-version programming* disappears in a parallel environment. However, the efficiency may not be close to 1 and the speedup may not be close to the number of processors available because of the time needed for communication between versions and the *voter*, which are running on different processors. This communication time is often several orders of magnitude longer than the times for operations on a single processor.

It is unlikely that any implementation of *N-version programming* with a high level of communication between the versions and the *voter* will ever have an efficiency much greater than 0.5. The only way to get a more efficient *N-version programming* system is to reduce the number of communications.

We ran our experiment on the same 12 files reported in [3] and found relatively low efficiencies. There are some unresolved questions about whether to count the time for loading Ada code on nodes as part of the execution time (by analogy with the time for dynamic instantiation of an Ada task) or ignoring it; we expect to return to these topics in future work.

3. Fault-tolerance of Message Passing and the Ada Rendezvous

The Ada rendezvous mechanism provides a high level of synchronization between tasks. When incorporated with the Ada exception detection and handling features, a programmer has a considerable amount of ability to prevent the catastrophic failure of a system. It is fairly simple to design the voter task so that it can continue to function in the event of a faulty version and can easily create a new task to use in the place of the faulty version. The Ada language provides a high level of control over failures and a highly fault-tolerant system can be designed.

The message passing system of the Intel hypercube is less fault-tolerant than the Ada mechanisms indicated above. The message passing subsystem is based in large part on the message passing facilities available in UNIX. This subsystem and the underlying hardware are themselves quite reliable. However, it requires the use of a fixed number of buffers of various sizes and its performance can be somewhat problematical if many large messages arrive and the nodes have small memory capacity. (Recall that there were a maximum of 634 buffers available for messages of 100 bytes or less.)

A more serious problem with the message passing is the lack of control via Ada language features. A failure of a message to be received can be communicated to the sender by noting the lack of an acknowledgement. However, the communication of this failure to an exception detection is not smooth because of the differences between the message-based NX/2 operating system and the Ada language. In particular, we must do a considerable amount of testing of the propagation of message passing errors to the sender or receiver tasks before we can be certain of the ability to detect faults.

In addition, the detection of hardware faults by Ada programs in this environment is still an open research question.

We have not yet made a formal study of the nature of the propagation of exceptional behavior between tasks on a single node and propagation of exceptions across other nodes of the hypercube. This is a major research direction for us.

4. Ease of Programming in a Parallel Environment

The implementation of *N-version programming* in a single processor environment involves several actions:

- creating the multiple versions by different people

- forming the versions, which were created as sequential algorithms, into tasks

- creation of the voter

- creation of the main program, including creation of tasks for the versions and voter and making decisions about what constitutes a failure and what actions are needed in the case of a failure.

- arrangements for I/O, including file access

Other activities may also be needed in special cases.

Some of these activities are not necessary in a parallel environment. For example, since the versions will probably be running on their own processor, we don't need to form the versions into tasks. Thus the individual versions can be used as is. The main program does not need to create tasks; instead, it needs to send pre-compiled executable files for the versions and the voter to the various processors. This is somewhat simpler than in the sequential version.

This simplicity is evident in the size of the source code needed for the various versions. The entire system grew from a sequential machine *N-version programming* system (with a maximum of 6 versions) of 8517 lines to a hypercube *N-version programming* system (also with a maximum of 6 versions) of 10221 lines, with the voter decreasing from 613 to 379 lines. Many of the new lines were comments so that relatively few new lines of source code needed to be added. Although we did not choose to do so because we already had formed the versions into tasks, the versions could have been run as sequential programs on the nodes directly. If we had done so, the parallel *N-version programming* system would have had fewer lines of code than the system on the sequential machines.

The creation of a main program is very simple; it only needs to load executable files onto the various nodes.

The programming complexity is increased by only one aspect: we need to synchronize the voter to get responses. This is more difficult in this hybrid system than it is on a pure Ada system in which the rendezvous mechanism is available.

The facilities for I/O are somewhat complex in that most I/O devices are inherently sequential devices and do not support multiple access. The Intel hypercube allows the use of what is called the *Concurrent File System*, which is a software organization of the attached disk that allows the use of multiple file pointers in a manner that is transparent to the user. More advanced (and expensive) hypercubes available from Intel (and others) allow the use of dedicated "I/O nodes", which are processors that are able to perform disk I/O rapidly and of course concurrently. Without the use of these concurrent I/O facilities, the file is accessed from the host node and communication to the other nodes is done by messages.

5. Ada and N-version Programming on Other Parallel Systems

Most parallel computers fall into one of three categories: distributed memory, such as the Intel hypercube; shared memory, such as the Alliant; and systems that have fairly large distributed memory and a small amount of shared memory.

Ada implementations on other distributed memory parallel computers will generally have the same advantages and disadvantages as on the Intel hypercube. This is due to the difficulties expected in allowing a rendezvous between Ada code running on different nodes. Thus our results will hold for most other systems of this type. In particular, the ease of having versions run without having to be made into tasks and the lack of a rendezvous between tasks running on different processors will occur.

The situation is more complex for shared memory parallel computers, however. It is much easier to have communication between different tasks if they can share a common memory. It is not clear what is needed to have

an efficient Ada rendezvous mechanism in this case.

6. Conclusions

The use of a parallel system for *N-version programming* allows a considerable reduction in the time for execution of multiple versions but retains the overhead of having communicating multiple processes. While the efficiency, at least as measured by the standard definitions of parallel programming, is low, there is an opportunity for improved performance.

The use of parallel computations also allows the direct use of separately coded versions rather than reforming them into Ada tasks. This is a simpler environment for *N-version programming*.

The hypercube implementation of Ada causes some problems, however. The unavailability of a rendezvous between software executing on different processors makes synchronization more complex and raises some issues about the fault-tolerance of such a system. Also, the relatively long time needed for communication between nodes suggests that *N-version programming* will be efficient only if there are few breakpoints for communication to the voter.

References

1. Avizienis, A. N-Version Approach to Fault-Tolerant Software, *IEEE Trans. Software Engr.* SE-11 (1985), 1491-1501.
2. Coleman, Don M., and Ronald J. Leach, "Performance Issues in C Language Fault-Tolerant Software", *Computer Languages*, vol 14, No. 1 (1989), 1-9.
3. Leach, R. J., and D. M. Coleman, N-version Programming Using the Ada Tasking Model, *Proceedings of the Ninth Annual National Conference on Ada Technology*, March 4-7, 1991, Washington, DC, 135-141.
4. Leach, R. J. Ada Exceptions and Fault-Tolerance, *Proceedings of the Eighth Annual National Conference on Ada Technology*, March, 1990, Atlanta, GA, 338-343.
5. Luckham, D. C., and W. Polak. *ADA EXCEPTIONS: Specification and Proof*

Techniques Advanced Research Projects Agency of the Department of Defense under Contract MDA 903 - 80 - C -0159 and Rome Air Development Center under Contract F 30602 - 80 - C - 0022.

6. Randell, B., System Structure for Software Fault Tolerance, *IEEE Trans. Software Engr.* SE-1 (1975), 220-232.
7. Rochkind, M. J., *Advanced UNIX Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
8. United States Department of Defense, *Reference Manual For The Ada Programming Language ANSI/MIL-STD 1815A*, Secretary of Defense, Research and Engineering, Washington, D.C., 1983.

Acknowledgements

Research of Ronald J. Leach was partially supported by the U.S. Army Research Office under grant number DAAL03-89-G-100 and by Wright Laboratories under contract number F33615-91-C-1758.

Research of Don. M. Coleman was partially supported by Wright Laboratories under contract number F33615-91-C-1758.

About the Authors

Ronald J. Leach is a Professor in the Department of Systems & Computer Science at Howard University. His research interests include concurrent and parallel programming, especially in Ada; software engineering; operating systems; and fault-tolerant, real-time programming. He is a former Program Committee Chair for this conference. He is the author or co-author of more than 35 research papers. He has a PhD in Mathematics from the University of Maryland at College park and a MS degree in Computer Science from Johns Hopkins University.

Don M. Coleman is Professor and Head of the Department of Systems & Computer Science at Howard University. His research interests include concurrent and parallel programming, especially in Ada; software engineering; systems engineering; and fault-tolerant, real-time pro-

gramming. He has been the Principal Investigator on many research projects including current projects in fault-tolerance, program translation, and transitioning to Ada.

Lu P. Tan is a graduate of Howard University, where he was awarded the MS degree in both Computer Science and Mechanical Engineering. He has worked at Howard University on two research projects and has been a consultant on a contract to IBM and NIST

Arabic OOD Methodology for use of Ada in the Arab World

by

Jagdish C. Agrawal and Abdullah M. Al-Dhelaan¹

Computer Science Department

King Saud University

P. O. Box 5117

Riyadh-11543

Kingdom of Saudi Arabia

ABSTRACT

In the literature, it has successfully been demonstrated that the principles of software engineering can be practiced successfully while using OOD methodology for developing systems for implementation in Ada. OOD methodology proposed by Abbott [1] and Booch [2, 3] begins with a natural language, i.e., English, which makes the transition from requirements expressed in the natural language English to specifications and design using Abbott and Booch's OOD methodology. However, in the non-English speaking world, such a transition requires a front end transition for requirements from one's native language to English.

Translation of requirements from one natural language to another can be additional cause of errors in the requirements for several reasons. First, the ambiguities contributed by a natural language will probably about double. Second, the inexactness of translation because of lack of one-to-one mapping from one natural language to another is likely to contribute additional errors. We felt that if one could develop a methodology similar to that of Abbott and Booch which begins with the natural language Arabic, it will have several benefits. First, the requirements from Arabic could directly be mapped to specifications expressed in Ada using such a methodology. Second, it will stimulate building of automated tools for Arabic version of OOD methodology. It will make Ada more attractive to the Arabic speaking world. This paper proposes just such a methodology.

¹Currently: Deputy Director General, National Information Center, Ministry of Interior, Kingdom of Saudi Arabia

1.0 INTRODUCTION

Object Oriented Design (OOD) methodology lets an application system designer use the strong software engineering support of Ada programming language. With such a methodology, a system designer *need not map the problem domain into predefined data and control structures present in the implementation language*. Using OOD methodology, the developer can create his own functional abstractions as well as abstract data types more suited to the problem, mapping the real world problem into more natural solution space that has virtually unlimited range of abstractions and abstract data types. Also, OOD methodology strongly helps in the implementation of four of the software engineering principles -- abstraction, localization, information hiding, modularity -- through encapsulation of data objects and their methods in Ada units called packages. While Ada provides a strong tool for

the practice of software engineering, OOD methodology provides strong engineering support for the use of that tool in application system development.

2.0 OOD – English Description to Design

OOD methodology proposed by Abbott [1] and Booch [2, 3] makes a strong case that OOD begins with a natural language, i.e., English. By using several design problems, Booch has successfully demonstrated that the principles of software engineering [4] can be practiced successfully while using such OOD methodology for developing systems for implementation in Ada. EVB Software Engineering, Inc. [5] further refined the work of Abbott and Booch, evolving a step by step method for OOD. However it strongly depends on the use of English language for three important steps in this OOD methodology:

- (1) The definition of the problem,
- (2) Description of an informal strategy for software solution of the real world problem, and
- (3) Identification of the objects, their attributes and the operations applicable to these objects.

3.0 Technology Transfer needs of the Arab World

Preceding section has described an excellent news for the Ada customers who prefer the English language for the expression of the requirements of their problem. However, for other customers in the world who like the strong software engineering support of Ada for their future systems, but prefer to express the requirements of their problem in a language other than English, the above mentioned OOD methodology is of little use.

Technology transfer of methods that are heavily dependent on a particular language (i.e., English) to the

sovereign nations where the national language is much different from the language on which the methods depend, creates much challenge for the trainers of the method! The technology transfer becomes easier if necessary modifications to the method are made so that it can be expressed in the preferred language of the nations where such methods and tools are sold. While some research efforts in the area of multilingual computer systems have taken place [e.g., 6, 7, 8, 9], little work has appeared in the literature on modifications to software development methods to make them easily usable by the customers in the non-English speaking world. Heavy dependence of OOD methodology is an area where such modifications are possible. In this paper, we are demonstrating necessary modifications to OOD, to adapt it to the entire Arabic speaking world, most of which were strong allies and supporters of the leader country that produced Ada. We feel, therefore, that this paper is within full spirit and

theme of the 10th ANCOAT: "Ada in Context: Economy, Geopolitics and Technology."

The Arab world has already built bilingual microcomputer systems [6]. We are proposing to extend the OOD methodology so that it begins with customer's requirements expressed in Arabic. This paper describes the theoretical aspects of our OOD methodology that begins with the requirements expressed in Arabic. The paper will demonstrate our methodology with examples.

4.0 Introduction to Arabic OOD Methodology

Our Arabic OOD methodology accepts requirements expressed in Arabic, analyzes them and prepares:

- (1) Problem definition.
- (2) Informal strategy for software realization of the problem.

- (3) Identification of "Issm" (nouns) and "Dhamir" (pronouns).

Certain "Issm" are not likely to be realized in the software solution because they are from environment external to the software. "Sofat" (adjectives) corresponding to the Issm/Dhamir are searched for possible attributes. Context semantics is used to determine the noun categories. The category of common noun is used to identify classes of objects, and the category of proper nouns is used to identify instances of classes previously identified. This work is used to prepare an object table. The table lists Issm/Dhamir, and whether each one falls within the software space of external to it.

- (4) Identification of "Amal" (actions or operations) that change the "halab" (state) of the Issm. This step helps in describing the "Tabakah" (classes). This is done by identifying "Fiel" (verbs) and "Zarf" (adverbs) along with

their relationship to the objects in the object table.

The natural language used for all the steps above is Arabic. Steps (3) and (4) require the designer to parse the informal strategy prepared from the requirements and determine which Issm/Dhamir (noun/pronoun) changes the Tabakah (state) by the action of the Fiel (verb). Identification of the states of an object, and actions that cause the changes in the state assist the designer in converting his objects into finite state machines and encapsulating the objects and methods/procedures within the class of such objects. Ada provides a fine mechanism of program units called package specification and package body. Steps 2, 3, and 4 are refined and repeated, lowering the level of abstraction, each time providing more and more implementation detail, until the software solution is ready for ready implementation in Ada.

The methodology we present in this paper can be adapted to other natural languages that have syntax closer to that of Arabic. Specific application of our methodology with specific examples falls in the category of proprietary information and intellectual property of authors, which we would like to protect from becoming part of the public domain, hence it is omitted in the paper.

REFERENCES

1. Abbott, R., "Program Design by Informal English Descriptions," Communications of the ACM, 1983.
2. Booch, G., "Object-Oriented Development," IEEE Transactions on Software Engineering, March 1986.
3. Booch, G., Software Engineering with Ada, The Benjamin / Cummings Publishing Company, Inc., Menlo Park, CA, 1986.
4. Ross, D.T., Goodenough, J.B., Irvine, C.A., "Software Engineering:

Process, Principles, and Goals," IEEE Computer, May 1975.

5. Object Oriented Design Handbook, EVB Software Engineering, Inc., Rockville, MD, 1986.

6. Tayli, M., and Al-Salamah, A., "Building Bilingual Microcomputer Systems," Communications of the ACM, volume 33, no. 5, May 1990.

7. Sibley, E.H., "Alphabets and Languages," Communications of the ACM, volume 33, No. 5, May 1990, pp. 488-489.

8. Jinan Qiao et al, "Six Digit Coding Method," Communications of the ACM, volume 33, no. 5, May 1990, pp. 491-494.

9. Raman, S., and Alwar, N., "An AI-Based Approach to Machine Translation In Indian Languages," Communications of the ACM, vol. 33, no. 5, May 1990, pp. 521-527.

DISA'S ROLE IN THE CENTER FOR INFORMATION MANAGEMENT

Presenter: Mr. Peter M. Fonash, Center for Information Management

CORPORATE INFORMATION MANAGEMENT (CIM) PANEL

Moderator: Dr. Kurt Fischer, OASD-C31

Panelists: LtCol Ralph H Anzelmo, US Marine Corps

COL James Voeltz, USAISSC

Col Fred Mellor, US Air Force

CAPT Kathleen Laughten, US Navy

ENGINEERING "UNBOUNDED" REUSABLE ADA GENERICS

Joseph E. Hollingsworth
Bruce W. Weide

Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, Ohio 43210-1277

holly@cis.ohio-state.edu, (614) 292-5813
weide@cis.ohio-state.edu, (614) 232-1517

Abstract

Most current programming languages (including Ada) provide some means of allowing the programmer to dynamically allocate and deallocate heap storage. This permits construction of "unbounded" abstract data types, e.g., stacks, queues, one-way lists, etc. Unfortunately, the addition of dynamically allocated storage to the implementation of abstract data types is a complicated business. Unless special care is taken, it can lead to problems of storage leaks, dangling references, unwanted aliasing, and unexpected lengthy execution times (due to storage allocation and reclamation), among others. We propose a specific discipline for avoiding these problems.

1. Introduction

Section 4.2 of the Ada 9X Requirements¹ recognizes the following storage management problems associated with abstract data types (ADTs) implemented using dynamically allocated data structures ("unbounded" ADTs):

- Problem 1. Currently the implementer of an ADT has no sure way of regaining control over dynamically allocated storage. However, as noted in the Ada 9X Requirements¹, p. 19: "...the programmer should be able to gain control whenever storage is allocated and whenever a scope is deactivated."
- Problem 2. There is a proliferation of unbounded ADTs providing the same functionality, differing only in their storage management scheme. The need for different schemes is also recognized in the Ada 9X Requirements¹, p. 19: "...the ability to provide specialized storage management algorithms is often essential when tuning an application's performance." This is already happening in practice. Booch⁴ typically provides two versions for each unbounded type, e.g., `Stack_Sequential_Unbounded_Managed_Noniterator` and `Stack_Sequential_Unbounded_Unmanaged_Noniterator`. These differ only in their storage management scheme.
- Problem 3. The client of an unbounded ADT typically has little control over the allocation and reclamation process, but needs this control. Again, this point is made in the Ada 9X Requirements¹, pp. 19-20: "For time-critical applications, storage

This material is based upon work supported by the National Science Foundation Grant No. CCR-9111892.

allocation and reclamation actions must occur at predictable times and must be accomplished in a bounded amount of time."

This paper offers a discipline for designing unbounded ADTs based on a coherent set of engineering principles. By adhering to this discipline, an Ada software designer can develop a large class of unbounded ADTs that do not suffer from the storage management problems listed above.

One of the principles presented in this paper is related to work by Booch⁴ and Musser and Stepanov¹³ in that it involves an encapsulation for storage management. It is different from their work in that the encapsulation imposes stricter control over the client's use of the allocated storage. Through this strict encapsulation, problems 2 and 3 can be addressed. Rosen¹⁵ proposes an abstraction with stricter control than Booch or Musser and Stepanov, but this encapsulation is not used to build linked structures, as is done here. Work by Sherman¹⁶, Muralidharan¹², and Baker³ also addresses Problem 1.

The paper is organized as follows. Section 2 introduces a discipline for designing unbounded ADTs with acyclic linked representations. Section 3 introduces *Nilpotent_Template*, a package that encapsulates storage management for such ADTs. Section 4 demonstrates the use of *Nilpotent_Template* by two different clients, and Section 5 discusses alternative implementations for *Nilpotent_Template*. Section 6 extends the *Nilpotent_Template* concept to deal with tree and DAG structures, and Section 7 presents our conclusions.

2. Discipline for Designing Unbounded ADTs

Henceforth, when we discuss "generic packages," we mean Ada generic packages that export unbounded ADTs and operations to manipulate them. The main objective of this paper is to present a formally-based design discipline for such generic packages that is based on five engineering principles. The point is to show why the whole is greater than the sum of the parts (i.e., the engineering principles standing separately).

The individual engineering principles are:

- Principle 1. A generic package must export an initialize and a finalize operation for each exported type, to be called by the client on each variable of that type upon entry to and exit from its scope, respectively.
- Principle 2. A generic package must export only limited private types. A data movement operation that properly enforces the abstraction must be provided for variables of each exported type (e.g., Copy or Swap⁷).
- Principle 3. A generic package must export package initialize and finalize operations to be used by the client for each instance of the package upon entry to and exit from its scope, respectively.
- Principle 4. Storage management must be encapsulated in its own generic package. This abstraction must enforce strict control over access to allocated storage, and must admit a variety of possible implementations.
- Principle 5. A generic package must import storage management operations through generic parameters. These operations should be those provided by the generic package described in Principle 4.

Principle 1 is not new (see Sherman¹⁶), but it is necessary because the finalize operation is the only way in which the unbounded generic package can regain control of the storage allocated to variables of any type exported by the package. Principle 1 addresses problem 1.

Exporting each type as limited private, required by Principle 2, has been suggested by many, e.g., Hibbard⁸, Booch⁴, and Edwards⁵. A data movement operation that properly enforces the abstraction does so without creating an alias; this is required for modular verification of Ada generics (see Ernst⁶). For efficiency reasons⁷ we choose the Swap operation in our designs, but the traditional Copy operation would also suffice.

Principle 3 requires a package finalization operation so that storage allocated to package-level

variables can be reclaimed prior to the deactivation of the package (i.e., prior to leaving the scope in which the package was instantiated). Although there is a syntactic slot provided by Ada for optional package initialization, an explicit procedure is required of all packages by our discipline. This produces symmetric, consistent and uniform interfaces. Principle 3 also addresses problem 1.

Principle 4 is followed in part both by Booch⁴ and by Musser and Stepanov¹³, but their encapsulation is not strict enough. In general, what is meant by "strict" is that there is no uncontrolled aliasing of the allocated storage, i.e., all copies of pointers are made by operations provided by the abstraction. This strict control allows the implementation to use alternatives for storage reclamation that otherwise would not be available (see Section 5). These alternatives address problems 2 and 3 in that the implementation of the abstract storage management package can be "tuned" to the client's needs without changes to the client (except in the package instantiation).

Principle 5 attacks the proliferation of different versions of functionally similar unbounded ADTs by making them parametric in the storage management scheme. Also, it gives the client control over major performance factors (storage allocation and reclamation), which is crucial if a high level of ADT reuse is to be achieved, especially in real-time systems². Principle 5 addresses problems 2 and 3.

To understand the details of the discipline, it is necessary to work at five different levels, or layers, of software. Principle 4, encapsulating storage management, is concerned with the lowest two levels, the storage management abstraction (see Section 3) and its implementation (Section 5). Principles 1, 2, 3, and 5 are concerned with the next two higher levels, a generic package exporting an unbounded ADT and its implementation, which is based on the storage management abstraction (Section 4). At the highest level is the client of this generic package, whose use of the generic package is discussed in Principles 1, 2, and 3 (Section 4). This view of the discipline follows the 3C (concept, content, and context) model of software structure (see Latour¹¹).

3. An Abstraction for Acyclic Pointer Structures

In this section we introduce Nilpotent_Template, an abstraction for acyclic pointer structures. Understanding this abstraction is paramount for understanding the discipline. Why is this abstraction so important? It allows us to get pointers right once and for all, eliminating the troublesome details associated with using programming language pointers when implementing acyclic linked structures. Furthermore, if a generic package is parameterized by Nilpotent_Template, its performance can be tuned by the client with respect to storage management. The generic package stays the same, while the client simply selects and instantiates an alternative implementation of Nilpotent_Template (see Section 5 for a discussion of alternative implementations).

Experience shows that the abstraction may be difficult to understand, so we begin with a simple example, working up from there. Suppose one is developing a program that requires a singly linked list for storing a character string. There is one problem: the programming language being used does not support pointers. What can be done? Simulate the pointers using an array for storing the characters while maintaining a parallel array for storing the simulated pointer link to the next character. This is common in FORTRAN programs, and is taught in some introductory data structures courses and standard texts (see Horowitz¹⁰). For example, suppose we have the list (a, x, r). An implementation using simulated pointers and parallel arrays might look like the following:

	head		tail		
	1		3		
label	a	x	r		...
target	2	3	0	0	...
	1	2	3	4	...

Figure 1 — Simulated pointers using parallel arrays.

To access the first item in the list, "a," use the value stored in the variable head to index into the label array. To find the next item in the list, simply index into the target array and use the value stored there as the index into the label array. The end of the list is reached when the value in the target array is zero.

Abstracting from this implementation leads to a specification for the desired generic abstract data type. There are actually two mappings at work in this example, a mapping from integers to characters, and a mapping from integers to integers. These mappings can be viewed as mathematical functions, label and target, having the following mathematical form:

label: integer \rightarrow Item
target: integer \rightarrow integer

By definition label and target are both total functions that form part of a complete mathematical model of the simulated pointer structure. To remain within the page limit, in examples we show only ordered pairs for which the domain value is of interest. For the above example, the functions have the following values:

label = {(1, a), (2, x), (3, r)}
target = {(1, 2), (2, 3), (3, 0)}

The Abstraction in the Form of an Ada Generic Package

The abstraction Nilpotent_Template presented below (see Figure 2) is based on the two functions label and target. It exports a program type called Position (modeled by the integers used as the domains of the functions), and it exports operations for manipulating the functions (i.e., for evaluating and changing them). The package is parameterized by the type Item. The specification has three parts: Ada code; mathematical specifications (in Ada comments beginning with --!); and English explanation (in Ada comments beginning with only --). The mathematical specifications are similar to those found in Pittel¹⁴, with modifications to facilitate the presentation and to correspond with the Ada implementation. Upon first reading, think of the above example and use the associated

explanations as an aid to understanding the specification. Then move on to the example client program found in the next section, referring back to the specification when necessary.

We have some experience introducing Nilpotent_Template to programmers in the classroom*. The students' first encounter with Nilpotent_Template was difficult. However, with some explanation of the specs along with an example similar to the one found in the next section, the students became comfortable with the abstraction and easily were able to use it to implement two different linked structure packages (queue and one-way list).

4. A Client of the Nilpotent Template

This section demonstrates the use of the Nilpotent_Template introduced in the last section by providing two clients: an Ada procedure that instantiates Nilpotent_Template and an Ada generic package parameterized by Nilpotent_Template.

A Procedure to Create a Simple List

The procedure of Figure 3 creates the example list, (a, x, r), from Section 3. The reader should refer to the remainder of Figure 3 for three different views of the program's execution. Figure 3 has nine rows and four columns. The nine rows correspond with the nine lines in the program tagged with the comment "-- #." Column one contains the line number; column two contains an illustration of the abstract state; column three illustrates a simulated pointer representation; column four shows a standard representation using pointers and nodes with "next" fields.

* Eighteen graduate and upper-division undergraduate students, in a class called "Software Components Using Ada" (see Hollingsworth⁹).


```

--! concept Nilpotent_Template
--!
--!     conceptual context
--!
--!         generic
--!         conceptual parameters
--!
--!             type Item
--!                 type Item is limited private;
--!                 with procedure Initialize (x: in out Item);
--!                 with procedure Finalize (x: in out Item);
--!                 with procedure Swap (x1: in out Item; x2: in out Item);
--!
--!         mathematics
--!         math variables
--!
--!             used: integer
--!             label: function from integer to math[Item]
--!             target: function from integer to integer
-- Conceptually Nilpotent_Template maintains these three internal "state" variables. The exported
-- operations manipulate these variables as well as their actual parameters. Nilpotent_Template's
-- implementation (package body) is not obligated to represent these variables explicitly,
-- as they are mathematical abstractions, not Ada variables.
--!
--!             initially      "used = 0 and
--!                             for all i: integer (Item.init (label (i)) and (target (i) = 0))"
-- Conceptually Nilpotent_Template dispenses unused positions beginning at integer number one.
-- Positions are dispensed to the client via the operation Attach_Label (see below). Each time
-- a position is dispensed, the math variable used is incremented by one. There is no danger of
-- eventual overflow because used is a mathematical integer, not an Ada integer.
--!
--! package Nilpotent_Template is
--! interface
--!
--!     procedure Initialize_Package;
--!     procedure Finalize_Package;
--!
--!     type Position is modeled by integer
--!     exemplar p
--!     constraint      "p >= 0"
--!     initially      "p = 0 and used = #used and
--!                     label = #label and target = #target"
--!     finally        "used = #used and label = #label and target = #target"
--!     type Position is limited private;
--!     procedure Initialize (p: in out Position);
--!     procedure Finalize (p: in out Position);
--!     procedure Swap (p1: in out Position; p2: in out Position);
-- Conceptually the type Position is modeled by a mathematical integer. Every variable p of type
-- Position is initially 0. The Initialize operation for a Position variable p does not have
-- an effect on the Nilpotent_Template's internal mathematical variables used, label and target,
-- nor does the Finalize operation.
-- In the post-condition (ensures clause) of an operation, the '#' preceding a variable
-- indicates the value of the variable at the beginning of the operation. A variable without the
-- '#' stands for the value of the variable at the end of the operation. The '#' is not used in a
-- pre-condition (requires clause).
--!
--!     procedure Attach_Label (
--!         p: in out Position;
--!         x: in out Item
--!     );
--!         --' produces
--!         --' consumes

```

```

--!      ensures "used = #used + 1 and p = used and
--!      for all i: integer (i /= p implies label (i) = #label (i)) and
--!      label (p) = #x and
--!      target = #target"
--      Conceptually this operation allocates the next unused integer to be used as a Position value.
--      It alters the label function, mapping the new Position p to the Item x. The new Position's
--      target is 0 because target initially maps every integer to 0. The operation also consumes
--      x; i.e., x is changed to an initial value for the type Item.

      procedure Swap_Label (
        p: in out Position;          --! preserves
        x: in out Item               --! alters
      );
--!      requires "p /= 0"
--!      ensures "used = #used and
--!      for all i: integer (i /= p implies label (i) = #label (i)) and
--!      label (p) = #x and x = #label (p) and
--!      target = #target"
--      Conceptually this operation allows a client to change the label function at Position p and
--      simultaneously to obtain the former label at Position p, by swapping. Neither used nor target
--      is changed.

      procedure Apply_Target (
        p: in out Position          --! alters
      );
--!      requires "p /= 0"
--!      ensures "p = target (#p) and
--!      used = #used and label = #label and target = #target"
--      Conceptually this operation applies the target function to p and sets p to the value
--      produced by the application.

      procedure Change_Target (
        p1: in out Position;        --! preserves
        p2: in out Position        --! preserves
      );
--!      requires "p1 /= 0 and p1 /= p2 and
--!      there does not exist k :integer, (k ≥ 0 and (target^k(p2) = p1))
--!      ensures "used = #used and label = #label and
--!      for all i: integer (i /= p implies target (i) = #target (i)) and
--!      target(p1) = p2"
--      Conceptually this operation allows a client to alter the target function by changing target(p1),
--      i.e., p1 now maps to p2 under the target function. Neither used nor label is changed.
--      Note: The notation target^k(p2) denotes the iterated application of target to p2, k times. For
--      example, target^2(p2) = target(target(p2)). The requires clause must be met so that no
--      circular structures will be created! In other words, target is a nilpotent function; hence
--      the name of the package.

      procedure Copy (
        p1: in out Position;        --! preserves
        p2: in out Position        --! produces
      );
--!      ensures "p2 = p1 and used = #used and label = #label and target = #target"
--      Conceptually this operation allows a client to create a copy of a position p1. None of the
--      internal mathematical variables are changed. Note: This is similar to aliasing if pointers
--      were being used. However this is the only way in which a client can create an alias.
--      Assignment is not available for limited private types. Implementations of Nilpotent_Template
--      can take advantage of this situation so that dangling references and storage leaks are
--      never created.

      procedure Test_If_Equal (
        p1: in out Position;        --! preserves

```

```

                p2: in out Position;                --! preserves
                equal: in out Boolean                --! produces
            );
--! ensures "(equal iff p1 = p2) and
--! used = #used and label = #label and target = #target"
-- Conceptually this operation sets equal to True if and only if p1 and p2 are the same integer.

private
    type Position_Rep;
    type Position is access Position_Rep;
    end Nilpotent_Template;
--! end Nilpotent_Template

```

Figure 2 — Nilpotent_Template Specification

```

with Nilpotent_Template,
    Built_In_Types;
use Built_In_Types;
-- Package Built_In_Types provides Initialize,
-- Finalize and Swap operations for the built
-- in Ada types Boolean, Character, Integer
-- and Float.

procedure Example_List is
package Character_List_Facility is
    new Nilpotent_Template (
        Character,
        Initialize,
        Finalize,
        Swap);
use Character_List_Facility;
head, tail, new_tail: Position;
c: Character;
begin
    Initialize (head);
                                Initialize (tail);
                                Initialize (new_tail);
                                Initialize (c);                -- 1
                                c := 'a';
                                Attach_Label (head, c);        -- 2
                                Copy (head, tail);             -- 3
                                c := 'x';
                                Attach_Label (new_tail, c);    -- 4
                                Change_Target (tail, new_tail); -- 5
                                Apply_Target (tail);            -- 6
                                c := 'r';
                                Attach_Label (new_tail, c);    -- 7
                                Change_Target (tail, new_tail); -- 8
                                Apply_Target (tail);            -- 9
                                Finalize (c);
                                Finalize (new_tail);
                                Finalize (tail);
                                Finalize (head);
                                end Example_List;

```

Line #	Abstract State	Simulated Pointer Representation	Standard Pointer Representation																					
1	head = 0 tail = 0 new_tail = 0 used = 0 label: {} target: {}	<table><tr><td>head</td><td>tail</td><td>new_tail</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>label</td><td></td><td></td></tr><tr><td>target</td><td>0</td><td>0</td></tr></table>	head	tail	new_tail	0	0	0	label			target	0	0	<table><tr><td>head</td><td>tail</td><td>new_tail</td></tr><tr><td></td><td></td><td></td></tr></table>	head	tail	new_tail						
head	tail	new_tail																						
0	0	0																						
label																								
target	0	0																						
head	tail	new_tail																						
2	head = 1 tail = 0 new_tail = 0 used = 1 label: { (1, a) } target: { (1, 0) }	<table><tr><td>head</td><td>tail</td><td>new_tail</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>label</td><td>a</td><td></td></tr><tr><td>target</td><td>0</td><td>0</td></tr></table>	head	tail	new_tail	1	0	0	label	a		target	0	0	<table><tr><td>head</td><td>tail</td><td>new_tail</td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr></table>	head	tail	new_tail						
head	tail	new_tail																						
1	0	0																						
label	a																							
target	0	0																						
head	tail	new_tail																						

Figure 3 — Three views of the example program's execution (continued to next page).

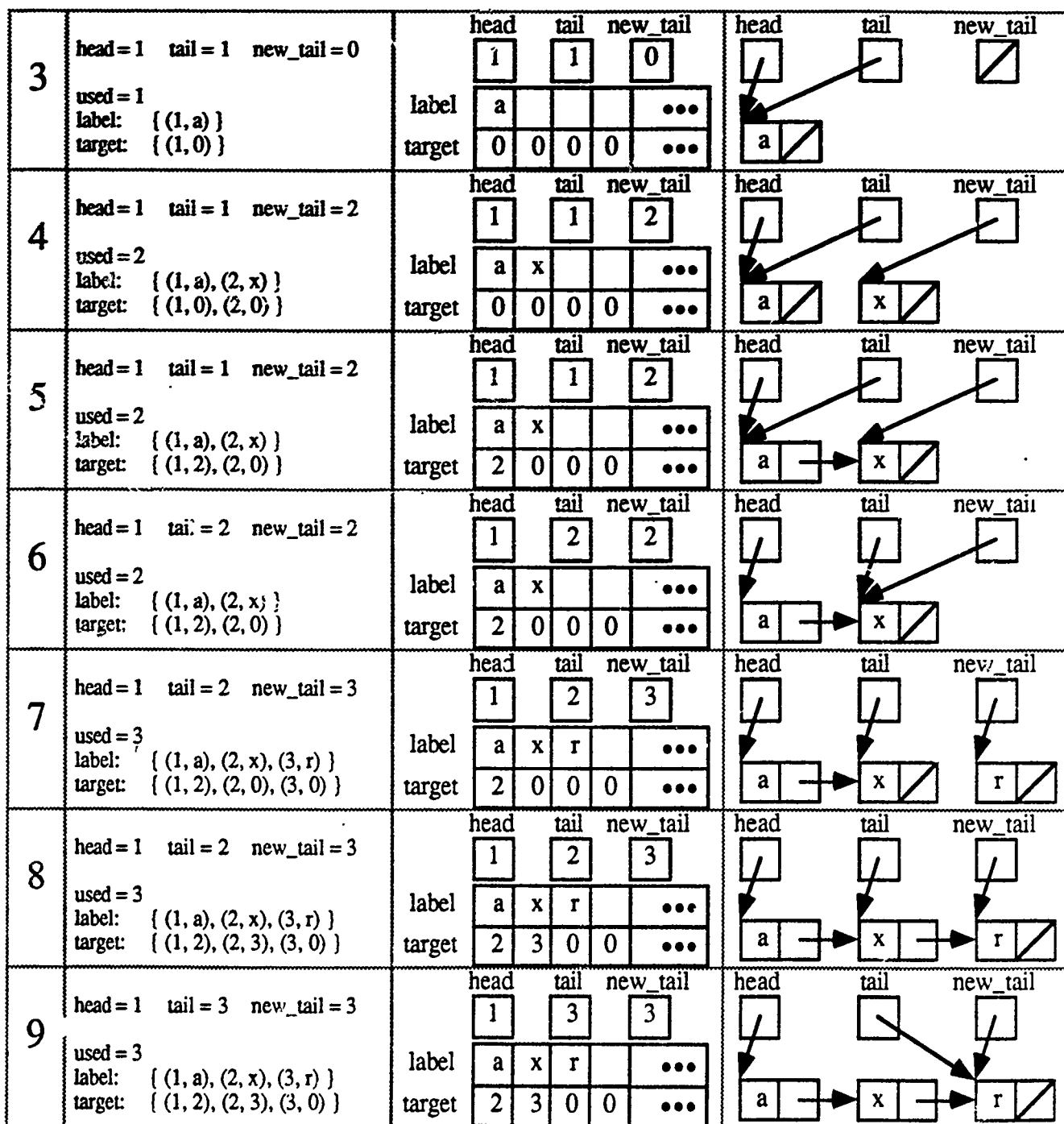


Figure 3 — Three views of the example program's execution.

The example program demonstrates how a client of the Nilpotent_Template can construct linked structures without directly using the programming language's pointer types. Additionally, this example illustrates that the programmer is

now able to reason about linked structures at an abstract level, as opposed to reasoning at the concrete or implementation level. Column three of Figure 3 demonstrates how a programmer reasons about linked structures in a language with-

cut pointers (e.g., FORTRAN); column four of Figure 3 demonstrates how a programmer working with a language that provides pointers reasons about linked structures (e.g., C, Pascal, or Ada). Column two, however, is how programmers should be reasoning about linked structures — at an abstract level.

Ada Generic Package Parameterized by Nilpotent Template

It is time to examine the discipline's next three levels: a generic package (Figure 4), its implementation based on Nilpotent_Template (Figure 5), and a client of the generic package (following Figure 5). For simplicity of presentation we have chosen stack. Abstractions such as queue, one-way list, and map illustrate the same points.

Stack_Template's specification follows the same format used for Nilpotent_Template, with formal mathematical specifications mixed with the Ada generic package specification. It has been engineered according to the principles outlined in Section 2. Conceptually, the type Stack is modeled as a mathematical string of Items (type Item is a parameter). Stack operations are specified in terms of mathematical string theory operations (e.g., concatenation). Close examination of the specification shows that the operations Initialize and Finalize satisfy Principle 1; Principle 2 is satisfied by the program type Stack being limited private, and by the Swap operation; Initialize_Package and Finalize_Package satisfy Principle 3; Principle 4 is satisfied by Nilpotent_Template itself. Nilpotent_Template's type and operations are generic parameters, satisfying Principle 5.

```
--! concept Stack_Template
--!     conceptual context
--!         uses
--!             STRING_THEORY_TEMPLATE

--!     generic
--!     conceptual parameters

--!         type Item
--!             type Item is limited private;
--!                 with procedure Initialize (x: in out Item);
--!                 with procedure Finalize (x: in out Item);
--!                 with procedure Swap (x1: in out Item; x2: in out Item);

--!     mathematics
--!     math facilities
--!         STRING_THEORY is STRING_THEORY_TEMPLATE (math[Item])

--!     realization context
--!     realization parameters

--!         facility Nilpotent_Facility is Nilpotent_Template (Item)
--!             type Position is limited private;
--!                 with procedure Initialize(p: in out Position);
--!                 with procedure Finalize(p: in out Position);
--!                 with procedure Swap(p1, p2: in out Position);
--!                 with procedure Attach_Label(p: in out Position;
--!                     x: in out Item);
--!                 with procedure Swap_Label(p: in out Position;
--!                     x: in out Item);
--!                 with procedure Apply_Target(p: in out Position);
--!                 with procedure Change_Target(p1, p2: in out Position);
--!                 with procedure Copy(p1, p2: in out Position);
--!                 with procedure Test_If_Equal(p1, p2: in out Position;
--!                     equal: in out Boolean);
```

```

package Stack_Template is
--! interface

    procedure Initialize_Package;
    procedure Finalize_Package;

--!
--! type Stack is modeled by STRING
--! exemplar s
--! initially "s = EMPTY"
    type Stack is limited private;
    procedure Initialize (s: in out Stack);
    procedure Finalize (s: in out Stack);
    procedure Swap (s1: in out Stack; s2: in out Stack);

    procedure Push (
        s: in out Stack;
        x: in out Item
        );
--! ensures "s = #s o #x"
        --! alters
        --! consumes

    procedure Pop (
        s: in out Stack;
        x: in out Item
        );
--! requires "s /= EMPTY"
--! ensures "#s = s o x"
        --! alters
        --! produces

    procedure Test_If_Empty (
        s: in out Stack;
        empty: in out Boolean
        );
--! ensures "empty iff s = EMPTY"
        --! preserves
        --! produces

private
    type Stack is new Position;
end Stack_Template;
--! and Stack_Template

```

Figure 4 — Generic Stack Specification

```

package body Stack_Template is

EMPTY_STACK_REP: Position;

procedure Initialize_Package is
begin
    Initialize (EMPTY_STACK_REP);
end Initialize_Package;

procedure Finalize_Package is
begin
    Finalize (EMPTY_STACK_REP);
end Finalize_Package;

```

```

procedure Initialize (s: in out Stack) is
begin
    Initialize (Position (s));
end Initialize;

procedure Finalize (s: in out Stack) is
begin
    Finalize (Position (s));
end Finalize;

procedure Swap (s1: in out Stack; s2: in out Stack) is
begin
    Swap (Position (s1), Position (s2));
end Swap;

procedure Push (s: in out Stack; x: in out Item) is
    new_top: Position;
begin
    Initialize (new_top);
    Attach_Label (new_top, x);
    Change_Target (new_top, Position (s));
    Swap (new_top, Position (s));
    Finalize (new_top);
end Push;

procedure Pop (s: in out Stack; x: in out Item) is
begin
    Swap_Label (Position (s), x);
    Apply_Target (Position (s));
end Pop;

procedure Test_If_Empty (s: in out Stack; empty: in out Boolean) is
begin
    Test_If_Equal (Position (s), EMPTY_STACK_REP, empty);
end Test_If_Empty;

end Stack_Template;

```

Figure 5 — Stack_Template Implementation

Below is a simple Stack_Client that is faithful to Principles 1, 3, and 5 of the discipline, by initializing and finalizing both the program variables and instantiated packages.

```

with Built_In_Types,
    Nilpotent_Template, Stack_Template;
use Built_In_Types;

```

```

procedure Stack_Client is

```

```

    package Nilpotent_Facility
    is new Nilpotent_Template (

```

```

        Character,
        Initialize,
        Finalize,
        Swap);

```

```

package Stack_Facility
is new Stack_Template (
    Character,
    Initialize,
    Finalize,
    Swap,
    Nilpotent_Facility.Position,
    Nilpotent_Facility.Initialize,
    Nilpotent_Facility.Finalize,

```

```

Nilpotent_Facility.Swap,
Nilpotent_Facility.Attach_Label,
Nilpotent_Facility.Swap_Label,
Nilpotent_Facility.Apply_Target,
Nilpotent_Facility.Change_Target,
Nilpotent_Facility.Copy,
Nilpotent_Facility.Test_If_Equal);

use Stack_Facility;

s1: Stack;
c: Character;

begin
  Nilpotent_Facility.Initialize_Package;
  Stack_Facility.Initialize_Package;
  Initialize (s1);
  Initialize (c);
  c := 'a';
  Push (s1, c);
  c := 'b';
  Push (s1, c);
  Finalize (c);
  Finalize (s1);
  Stack_Facility.Finalize_Package;
  Nilpotent_Facility.Finalize_Package;
end Stack_Client;

```

5. Nilpotent Template Implementation

Before discussing alternative implementations for Nilpotent_Template, we note the fundamental properties shared by all alternatives. Remember that Nilpotent_Template has been designed for building acyclic linked structures. Consequently, the precondition for Change_Target requires that no circularity be introduced into the linked structure that is under construction (see Figure 2). This requirement allows implementations to maintain reference counts for all dynamically allocated storage (see Weide¹⁷, Rosen¹⁵).

For example, when the client invokes Attach_Label, storage is allocated for a new position, and its reference count is set to one. The counts are updated by all operations of the Nilpotent_Template that change the values of Position variables or the target function. When the count reaches zero (note that the count can be decremented by a call to Finalize, Attach_Label, Change_Target, Apply_Target, or Copy) there are no positions that have access to the allocated

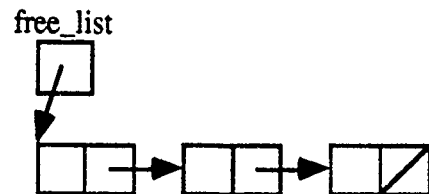
storage. At this time, the allocated storage may be reclaimed. Because of its interface, Nilpotent_Template has complete control over aliasing of allocated storage. If it did not, then the reference count system would break down.

This is also why circular linked structures are not allowed by the Nilpotent_Template as it stands. If they are allowed (by removing the requires clause of Change_Target), it is possible to build a structure where each piece of allocated storage has a reference count equal to one, but no position has access to the structure. To detect this situation, the implementation has to follow the target chain of a position whenever a reference count is decremented; otherwise storage leaks might occur. Following the target chain is potentially inefficient.

The Nilpotent_Template abstraction without the non-circularity constraint is useful for constructing circular structures, but does not admit an especially efficient implementation. On the other hand, it is no *less* efficient than using language-supplied primitives to allocate and deallocate storage, and it still supports abstract reasoning about client program behavior.

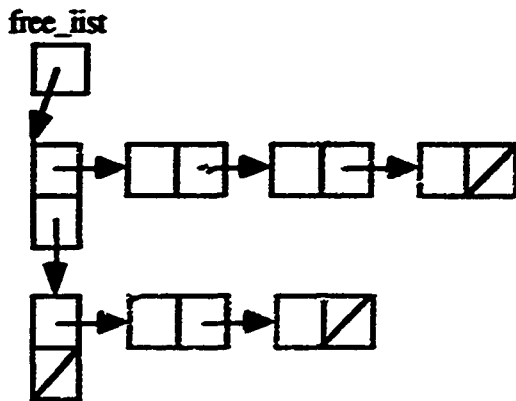
Here are four possible alternative strategies for storage reclamation:

- 1) Use the underlying run time system's garbage collector.
- 2) Use UNCHECKED_DEALLOCATION.
- 3) Maintain an internal free list of individual pieces of storage allocated by Attach_Label, as shown:



- 4) Maintain an internal free list of freed linked structures. These structures have

been constructed using `Attach_Label` and `Change_Target`, and have then been freed:



The first two strategies are straightforward, not warranting further discussion. The third strategy performs well when individual pieces of storage are freed (e.g., when a stack is popped), but suffers when an entire linked structure is freed. For example, if a client finalizes a stack of size N , this alternative's performance is necessarily linear in N . Why? Because it has to take each piece of storage off the stack and place it onto the free list. This might lead one to believe that the `Finalize` operation for a linked structure is inherently a linear-time operation, but it is not. The fourth strategy accommodates a constant time `Finalize` operation for linked structures of length N (see Weizenbaum¹⁵, Weide¹⁷). As stated above, this implementation maintains a free list of freed linked structures rather than a free list of individually allocated pieces of storage. The `Finalize` operation simply adds the entire size N linked structure to the free list, in constant time. With this storage management strategy, all `Nilpotent_Template` operations take constant time.

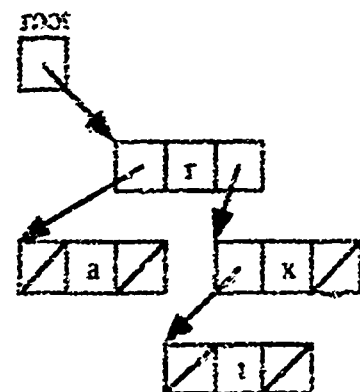
It is worth mentioning other factors that are relevant when the storage reclamation strategy maintains a free list. Unfortunately a full treatise on these factors would require another paper. For example, there might be a need to place an upper bound on the size of the free list; it might be to a client's advantage to populate the free list prior to the client requesting storage; and there is a question as to when is the best time to finalize the items in the data structure being reclaimed. That is, when the storage being reclaimed contains the only reference to some other large data

structure, when should that data structure be finalized?

By following the discipline outlined in this paper, `Nilpotent_Template` can be implemented using any of the above strategies. What's more, when generic packages are designed according to the discipline, they can be instantiated with any implementation of `Nilpotent_Template` and work equally well, achieving plug compatibility. Finally, when clients of a generic package faithfully adhere to the initialize/finalize requirements of the discipline, no storage leaks or dangling references can be created.

6. Extending the Nilpotent_Template Abstraction

With the `Nilpotent_Template` a software engineer can design and reason at an abstract level about singly linked structures, but not about multiply linked structures such as binary trees. To support multiply linked structures, `Nilpotent_Template` must be extended to support multiple target functions. The following example demonstrates a binary tree and its corresponding representation using simulated pointers and parallel arrays:



	root					
	1					
label	r	a	x	t	...	
target 1	2	0	4	0	...	
target 2	3	0	0	0	...	

The target function now has the following mathematical form:

target: integer \times integer \rightarrow integer

The target function for the above example is:

target = { ((1, 1), 2), ((1, 2), 0), ((1, 3), 4), ((1, 4), 0),
((2, 1), 3), ((2, 2), 0), ((2, 3), 0), ((2, 4), 0) }

N_Way_Nilpotent_Template has an additional generic parameter by which the client specifies the number of targets. As it turns out, *Nilpotent_Template* is a special case of *N_Way_Nilpotent_Template* (obtained by setting the number of targets to one). In practice we have implemented *N_Way_Nilpotent_Template*, and instantiated it with one target function to build singly linked structures and with two target functions to build binary trees.

7. Summary and Conclusion

We have introduced an engineering discipline for the construction of Ada generic packages that export "unbounded" ADTs, and their clients. The discipline addresses three problems:

- P1) regaining control over dynamically allocated storage;
- P2) the proliferation of unbounded ADTs differing only in their storage management scheme; and
- P3) no client control over an unbounded ADT's storage management scheme.

The discipline requires:

- R1) a totally encapsulated storage management module;
- R2) generic packages parameterized by and implemented with the storage management module;
- R3) total encapsulation of all types, including a data movement operation that properly enforces the abstraction;
- R4) initialization and finalization operations for all types and packages;
- R5) faithful use of initialize/finalize operations by all clients.

Problem P1 is addressed by requirements R3, R4 and R5. Problems P2 and P3 are addressed by R1 and R2.

The discipline, when properly applied, actually goes farther. It guarantees that: no storage leaks or dangling references can be created; unbounded generics are plug compatible with respect to storage management (giving the client control at instantiation time over the storage management scheme employed); the storage manager can be implemented so that all of its operations execute in constant time, including the Finalize operation for linked structures of size *N*; the interface of unbounded generics is uniform and consistent (permitting compositions such as stacks of one-way lists through generic instantiation).

Straightforward, comprehensible solutions to problems such as storage management, client control over performance, plug compatibility, uniformity, consistency and composability are not easy to find. One cannot take a half-hearted approach to solving these problems, i.e., one cannot adopt principles 1, 3, and 5, for example, and hope to gain much. Only when we examine the entire picture and are willing to take a different approach at all levels, do we come up with a comprehensive discipline for solving these problems.

Acknowledgments

We are pleased to thank the members of the Reusable Software Research Group at The Ohio State University for their comments and many helpful discussions on the content of this paper.

References

1. *Ada 9X Project Report: Ada 9X Requirements*, Office of the Under Secretary of Defense for Acquisition, Washington, D.C., Dec. 1990.
2. Allen, D., et al., eds., "Catalog of Interface Features and Options for the Ada Runtime Environment," *Ada Letters*, Vol. 11, No. 8, Fall 1991, 11-2.
3. Baker, H., "Structured Programming with Limited Private Types in Ada: Nesting is

for the Soaring Eagles," *Ada Letters*, Vol. 9, No. 5, July/Aug. 1991, 79-90.

4. Booch, G., *Software Components with Ada*, Benjamin/Cummings, Menlo Park, CA, 1987.
5. Edwards, S., "An Approach for Constructing Reusable Software Components in Ada," Institute for Defense Analyses, Alexandria, VA, IDA Paper P-2378, 1990.
6. Ernst, G.W., Hookway, R.J., Menegay, J.A., and Ogden, W.F., "Modular Verification of Ada Generics," *Comp. Lang.*, Vol. 16, No. 3/4, 1991, 259-280.
7. Harms, D.E., and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Trans. on Software Eng.*, Vol. 17, No. 5, May 1991, 424-435.
8. Hibbard, P., Hisgen, A., Rosenberg, J., Shaw, M., and Sherman, M., *Studies in Ada Style*, Springer-Verlag, New York, 1983.
9. Hollingsworth, J.E., Weide, B.W., and Zweben, S.H., "Confessions of Some Used-Program Clients," *Proceedings 4th Annual Workshop on Software Reuse*, Herndon, VA, Nov. 1991.
10. Horowitz, E., and Sahni, S., *Fundamentals of Data Structures*, Computer Science Press, Inc., Rockville, Maryland, 1976.
11. Latour, L., Wheeler, T., and Frakes, W., "Descriptive and Predictive Aspects of the 3Cs Model: SETA1 Working Group Summary," *Third Annual Workshop: Methods and Tools for Reuse*, Syracuse Univ. CASE Center, Syracuse, NY, June 1990.
12. Muralidharan, S., and Weide, B.W., "Should Data Abstraction Be Violated to Enhance Software Reuse?," *Proceedings 8th Annual National Conference on Ada Technology*, ANCOST, Inc., Atlanta, GA, Mar. 1990, 515-524.
13. Musser, D., and Stepanov, A., *The Ada Generic Library: Linear List Processing*

Packages, Springer-Verlag, New York, 1989.

14. Pittel, T.S., *Pointers in RESOLVE: Specification and Implementation*, M.S. Thesis, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, June 1990.
15. Rosen, S.M., "Controlling Dynamic Objects in Large Ada Systems," *Ada Letters*, Vol. 7, No. 5, Sept./Oct. 1987, 79-92.
16. Sherman, M., Hisgen, A., and Rosenberg, J., "A Methodology for Programming Abstract Data Types in Ada," *Proceedings of the AdaTEC '82 Conference on Ada*, ACM, Arlington, VA, Oct. 1982.
17. Weide, B.W., "A New ADT and Its Applications in Implementing 'Linked' Structures," Technical report, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, OSU-CISRC-TR-86-3, Jan. 1986.
18. Weizenbaum, J., "Symmetric List Processor," *CACM*, Vol. 6, No. 9, Sept. 1963, 524 - 544.

Joe Hollingsworth holds an undergraduate degree from Indiana University and a master's degree from Purdue University. Before returning to school as a Ph.D. candidate at The Ohio State University, he worked at Texas Instruments. He has also consulted for Battelle Memorial Institute on issues of software design in Ada. In his recent research at OSU he has developed a compiler, linker, and run-time system for RESOLVE, and has worked on a set of engineering principles that can be used to develop generic reusable software components in Ada. His Internet address is holly@cis.ohio-state.edu, and his postal address is Department of Computer and Information Science, 2036 Neil Avenue Mall, Columbus, Ohio 43210.

Bruce W. Weide is Associate Professor of Computer and Information Science at The Ohio State University. He received his B.S.E.E. degree from the University of Toledo in 1974 and the

Ph.D. in Computer Science from Carnegie Mellon University in 1978. He has been at Ohio State since 1978. His research interests include various aspects of reusable software components and software engineering in general: software design-for-reuse, formal specification and verification, data structures and algorithms, and programming language issues. He has also published recently in the area of software support for real-time and embedded systems. His Internet address is weide@cis.ohio-state.edu, and his postal address is the same as shown above.

Intelligent Abstract Data Types

Robert A. Willis Jr.

Larry Morell

**Department of Computer Science
Hampton University
Hampton, VA 23668**

Internet Addresses

willis@unixvax.hamptonu.edu

morell@unixvax.hamptonu.edu

Introduction

This paper discusses the concept and implementation of Intelligent Abstract Data Types (IADT's). An IADT is an Abstract Data Type (ADT) that runs concurrently with its clients. An ADT is an encapsulated data type or object with an internal representation and set of operations which manipulate it. The rationale for extending the concept of an ADT is two-fold:

- a concurrently executing IADT allows many "client" tasks to share a resource in real-time.
- the IADT can perform internal functions when its clients do not require its services.

IADT's can be viewed as objects which can be "inherited" and embellished upon. This paper will detail an example which incorporates two low-level IADT's (different list implementations) into a high-level *List* IADT.

Using Ada

Of all the non-experimental languages we have examined, only Ada provides the features required to fully implement IADT's. Ada is one of the few languages which has safe and general concurrent features. These features are not experimental; they conform to the same consistent philosophy of block-structuring, strong typing, and sound software design principles found throughout Ada, and they are unambiguously specified. These are not features which were grafted onto a programming language, but rather designed as an integral part of Ada. Therefore, their syntax and usage is regular and consistent with all other features found in the language. Since Ada is a general purpose programming language, it is quite easy to integrate concurrent and sequential processing into a program allowing one to develop programs with a rational balance of concurrency as needed. Ada is particularly well adapted to concurrent program design and development because the modularity of its tasking mechanism sup-

ports object oriented design (OOD) quite well. Ada's encapsulation features allows developer's of IADT's to provide truly transparent service to using programs.

The Concept of IADT's

The basic premise is that in a programming language, such as Ada, which supports concurrency and true encapsulation, it is feasible to create independent or intelligent ADT's which execute concurrently with other tasks. This is a non-traditional approach in that programs which utilize data types such as linked lists, queues, trees, sets, and graphs are typically conceived and written as sequential programs. IADT's free the programmer to utilize concurrent algorithms when appropriate. Advantages still accrue if an IADT is used in a program whose body contains sequential instructions. Appropriate intelligence can be given to the IADT which will allow it to perform a variety of internal functions, modify its behavior, or even change its internal structure if necessary.

One example is the binary tree. Typical tree operations are Insert, Delete, Find, and Modify. These operations are most efficient when the tree is relatively balanced. An IADT tree maintains information regarding its state and will balance itself when necessary and it is not servicing its clients. A more intelligent IADT tree could perform other internal functions as well.

Another IADT could be a *List*. This *List* would have the intelligence to monitor its use and dynamically change its internal structure as required. If usage were light and not many searches were prevalent, the internal structure could change to a linked list, or a hash table if necessary; each internal structure would be a lower level IADT itself. These changes would take place concurrently with the normal operations of the using program and would also be transparent to it.

Architecture of an IADT

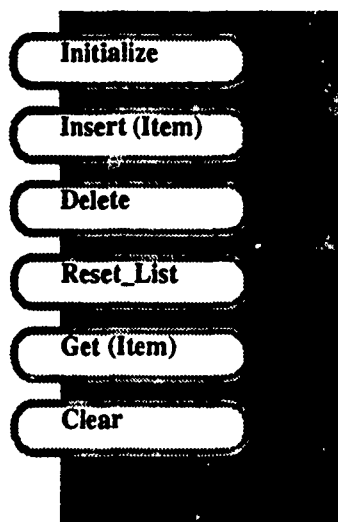
IADT's contain the following components:

- An external interface with which to communicate.
- An internal "intelligent module".
- Internal operations.

The External Interface

The external interface provides communication with using programs. All access to the data structure is through its interface. Figure 1 depicts possible operations available for a *List* IADT.

FIGURE 1. Architecture of an IADT: External Interface



The "Intelligent" Module

The "Intelligent" module (IM) does the following:

- Receives the request.
- Determines whether the request can be complied with.
- Notifies the requestor of its decision.
- Performs the task if possible.

The IM only delays the requestor long enough to decide and report (a rendezvous is implied). Completion of the request is performed concurrently with the requestor.

The IM is also responsible for maintaining the internal state of the data type. Various heuristics can be applied to ensure that the internal state is consistent with established criteria.

FIGURE 2. Architecture of an IADT: "Intelligent" Module

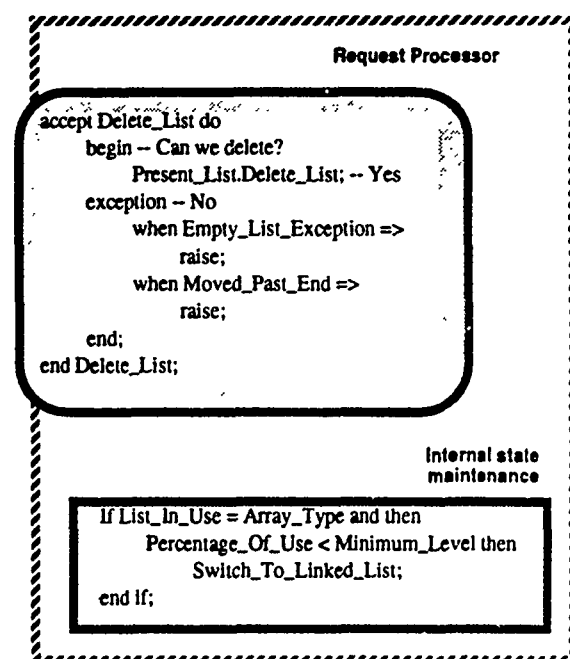


Figure 2 gives a partial view of the typical types of decisions which an IM must make. The request processor receives a request to delete the item at the present relative position in the list. It queries a low-level list IADT to determine if the task can be performed. If it can, no message is passed and the rendezvous is terminated. If it can't, the appropriate exception is reraised and propagated to the requestor.

When not servicing a client the IM is free to apply various heuristics to maintain a desired state. In figure 2 the IM checks to see if it is using an array to store data objects if it is and the number of items is very low then there is a switch to a linked list representation.

Internal Operations

The internal operations of an IADT are the same as the internal operations of any ADT. They are implemented as procedures and/or functions in the package body of the IADT.

IADT's In-depth

As stated earlier, two advantages of IADT's are

- **Increased parallelism:**
 - The ability to continue the execution of the caller while the IADT performs its operation, and
- **Dynamic reconfiguration:**
 - The ability of the IADT to perform bookkeeping operations off-line, with minimal impact to the client programs. In this section we describe the mechanisms of Ada that we used to achieve the above advantages.

Increased parallelism

The ability to have increased parallelism is inherent in the scheme we have established. All interaction with an IADT is conducted via rendezvous with a task we call the task manager. Upon accepting a rendezvous, the manager verifies the precondition of the requested operation. If the precondition is met, the client is released to continue execution, while the task performs the required operation in parallel. If another client request occurs before the previous request is finished, the tasking semantics ensure that the second request is blocked, pending completion of all prior requests. This is because each client request is completely processed before another rendezvous becomes possible. (Section **Future Work** of this paper suggests how this mode can be extended for increased parallelism.) The tasking implementation is hidden from the client program,

requiring no change in client code using a conventional ADT. The only restriction is that all operations must assume the form of procedure calls, a minor restriction considering the potential advantages.

Dynamic reconfiguration

The primary intelligence in IADT's is found in their ability to adjust their representation or behavior according to their run-time history. Adjustments may be desirable to improve space utilization or to decrease execution time. Programs frequently proceed through phases, in which different representations would be better. For example, while creating a database of student records it may be advantageous to use a linked representation to minimize the amount of time necessary for insertions and deletions. Once the student population has stabilized and retrievals dominate the executed operations, it may be advantageous to switch to a sorted array representation to save time and space.

Since our IADT's are implemented via tasking, they can handily accomplish the above scenarios. Each task body consists of a loop that first handles accepts and then performs necessary bookkeeping. The bookkeeping can include actions such as determining whether or not to switch representations or whether or not to perform different cleanup activities. In the text below we discuss the style and intent of the "intelligence" that can be incorporated into IADT's. By factoring such actions to a central location we improve the maintainability of our modules.

It should be noted that the principle of information hiding dictates clear division of labor between client and IADT when it comes to improving efficiency. Information hiding prevents the IADT from making assumptions about the client and the client module from making assumptions about the IADT, beyond that which is documented in the interface. In particular the client should not try to optimize

its performance based upon speculations of how the module is implemented. Neither should the IADT presume to know the space/time trade-off desired by the client. For example, it would be counterproductive for an IADT to optimize for space, when time is crucial.

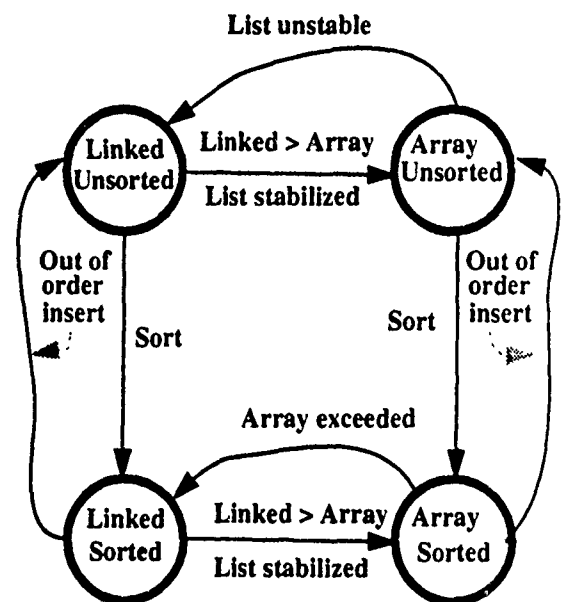
Information hiding encourages the use of assumptions over presumptions. It is therefore important to move presumptions about efficiency into the interface and document them properly, turning them into assumptions. We have therefore included as part of our interface an operation for the client to indicate the kinds of optimization that would be desirable. A call to `List_Efficiency (Desire)` will allow the client to specify `Desire` to be `SpaceOverTime` or `TimeOverSpace`. Processing this request will cause the module to optimize in the desired direction if possible.

How such optimization occurs is up to the IADT. In some situations the IADT can decide for itself on a particular matter of efficiency. For example suppose two representations of lists could be used, a linked (unbounded) representation and a array-based (bounded) representation. Suppose we wish to optimize for space. A linked representation is more efficient up to some threshold, beyond which the array-based implementation is more efficient. After crossing the threshold it is reasonable to switch representations. Similarly, if for a specified period of time the interaction has stabilized in such a way that a linked representation would be more efficient in time (say, when there are many insert/delete operations) it is perhaps appropriate to switch a linked implementation to an array-based implementation.

The key to the intelligence lies in the analysis of when a change of representation or algorithm is necessary. In our preliminary version of IADT's we have specified these changes as finite state machines in which the states denote a particular configuration of algorithms and data structures, and a transition is effected by

the occurrence of certain events. For our list example an event is exceeding a node threshold, exceeding the size of the bounded representation, and producing a particular mix of operations. Our starting state is one in which the list is stored using a pointer. When we reach a state in which the size of the list exceeds the size of array by 10 percent, we switch representation to an array. When the array is filled, we switch back to the linked representation. We keep track of whether or not the list is sorted, updating this information at every insert. As long as it is sorted, we use the more efficient algorithms for searching the list in each representation, switching to linear search when the list becomes unsorted. Finally if `TimeOverSpace` is specified, we switch to the array implementation as soon as the proportion of inserts and deletes falls below 10%.

FIGURE 3. Efficiency Transition Diagram



Future work

The key to maximizing concurrency is to allow the concurrent execution of IADT operation if at all possible. In our initial version operations for a single IADT must be run sequentially

since a second request to an IADT operation will be blocked until the prior one finishes. Such a blockage is not always required, however. For instance, in our list example, it is perfectly reasonable to allow certain combinations of operations to proceed simultaneously (e.g. `Get_List` and `List_Length`).

To solve this problem within the constraints of information hiding, we propose to implement some of the operations themselves as tasks and to augment the preconditions of each operation to check for the validity of the call under the constraints of the currently executing operation tasks. If this augmented precondition is satisfied, then the proposed operation can be executed concurrently with the other executing operations. Violation of the precondition means that processing is still progressing whose completion is essential before the requested operation can be performed.

These blockage preconditions are frequently dependent upon the particular implementation strategy, hence they can be dependent upon lower-level IADT's. The parent IADT can then determine its actions based upon the status of the lower level IADT's.

It is thus necessary not only for an IADT to determine its ability to progress, but also to enable any caller to determine its current ability.

One proposal then is to require every IADT designer to predetermine what operations can execute concurrently, and to provide a public boolean variable (for each operation) to indicate whether or not the operation is currently executable. Information hiding is preserved because how this determination is made is private to the IADT; the fact that it has been made is public. By inspecting these status variables of lower-level IADT's, a higher-level IADT can determine its ability to progress, and queue a request for service or delay the client, depending on the nature of the operation. For example, a sequence of inserts, moves, and deletes can be queued for a list without delaying the client, but

a subsequent get might have to be delayed.

The above discipline assumes there is but one client accessing the list. In the case of multiple clients, the situation is considerably more complicated by the IADT concurrency. Client implementors should not have to worry about concurrency of IADT's, focussing otherwise on ensuring that their sequences of calls to the IADT are issued in the required order. Thus, it is necessary for the IADT to ensure that sequence of issuance is the same as the execution sequence. To do so it may be necessary to use a priority queue with priority determined by the time stamp of the operation request.

Conclusions

We have introduced the concepts of intelligent abstract data types. An IADT offers the potential for increased concurrency and improved space/time trade-off management. We have discussed the features of Ada used to implement an initial version of this concept to provide limited concurrency and dynamic reconfiguration of the underlying representation. Possible extensions of this concept for improved concurrency and efficiency have been proposed.

Intelligent Abstract Data Types

Robert A. Willis Jr.

Larry Morell

**Department of Computer Science
Hampton University
Hampton, VA 23668**

Internet Addresses

willis@unixvax.hamptonu.edu

morell@unixvax.hamptonu.edu

Introduction

This paper discusses the concept and implementation of Intelligent Abstract Data Types (IADT's). An IADT is an Abstract Data Type (ADT) that runs concurrently with its clients. An ADT is an **encapsulated** data type or object with an internal representation and set of operations which manipulate it. The rationale for extending the concept of an ADT is two-fold:

- a concurrently executing IADT allows many "client" tasks to share a resource in real-time.
- the IADT can perform internal functions when its clients do not require its services.

IADT's can be viewed as objects which can be "inherited" and embellished upon. This paper will detail an example which incorporates two low-level IADT's (different list implementations) into a high -level *List* IADT.

Using Ada

Of all the non-experimental languages we have examined, only Ada provides the features required to fully implement IADT's. Ada is one of the few languages which has safe and general concurrent features. These features are not experimental; they conform to the same consistent philosophy of block-structuring, strong typing, and sound software design principles found throughout Ada, and they are unambiguously specified. These are not features which were grafted onto a programming language, but rather designed as an integral part of Ada. Therefore, their syntax and usage is regular and consistent with all other features found in the language. Since Ada is a general purpose programming language, it is quite easy to integrate concurrent and sequential processing into a program allowing one to develop programs with a rational balance of concurrency as needed. Ada is particularly well adapted to concurrent program design and development because the modularity of its tasking mechanism sup-

ports object oriented design (OOD) quite well. Ada's encapsulation features allows developer's of IADT's to provide truly transparent service to using programs.

The Concept of IADT's

The basic premise is that in a programming language, such as Ada, which supports concurrency and true encapsulation, it is feasible to create independent or intelligent ADT's which execute concurrently with other tasks. This is a non-traditional approach in that programs which utilize data types such as linked lists, queues, trees, sets, and graphs are typically conceived and written as sequential programs. IADT's free the programmer to utilize concurrent algorithms when appropriate. Advantages still accrue if an IADT is used in a program whose body contains sequential instructions. Appropriate intelligence can be given to the IADT which will allow it to perform a variety of internal functions, modify its behavior, or even change its internal structure if necessary.

One example is the binary tree. Typical tree operations are Insert, Delete, Find, and Modify. These operations are most efficient when the tree is relatively balanced. An IADT tree maintains information regarding its state and will balance itself when necessary and it is not servicing its clients. A more intelligent IADT tree could perform other internal functions as well.

Another IADT could be a *List*. This *List* would have the intelligence to monitor its use and dynamically change its internal structure as required. If usage were light and not many searches were prevalent, the internal structure could change to a linked list, or a hash table if necessary; each internal structure would be a lower level IADT itself. These changes would take place concurrently with the normal operations of the using program and would also be transparent to it.

Architecture of an IADT

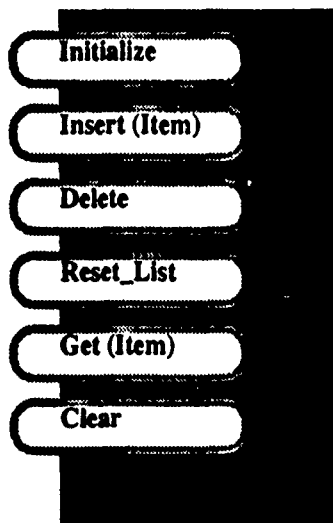
IADT's contain the following components:

- An external interface with which to communicate.
- An internal "intelligent module".
- Internal operations.

The External Interface

The external interface provides communication with using program. All access to the data structure is through its interface. Figure 1 depicts possible operations available for a *List* IADT.

FIGURE 1. Architecture of an IADT: External Interface



The "Intelligent" Module

The "Intelligent" module (IM) does the following:

- Receives the request.
- Determines whether the request can be complied with.
- Notifies the requestor of its decision.
- Performs the task if possible.

The IM only delays the requestor long enough to decide and report (a rendezvous is implied). Completion of the request is performed concurrently with the requestor.

The IM is also responsible for maintaining the internal state of the data type. Various heuristics can be applied to ensure that the internal state is consistent with established criteria.

FIGURE 2. Architecture of an IADT: "Intelligent" Module

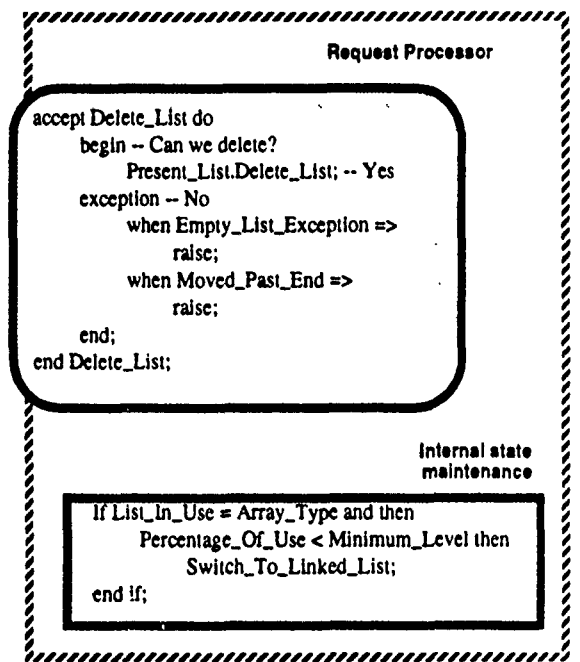


Figure 2 gives a partial view of the typical types of decisions which an IM must make. The request processor receives a request to delete the item at the present relative position in the list. It queries a low-level list IADT to determine if the task can be performed. If it can, no message is passed and the rendezvous is terminated. If it can't, the appropriate exception is reraised and propagated to the requestor.

When not servicing a client the IM is free to apply various heuristics to maintain a desired state. In figure 2 the IM checks to see if it is using an array to store data objects if it is and the number of items is very low then there is a switch to a linked list representation.

Internal Operations

The internal operations of an IADT are the same as the internal operations of any ADT. They are implemented as procedures and/or functions in the package body of the IADT.

IADT's In-depth

As stated earlier, two advantages of IADT's are

- Increased parallelism:
 - The ability to continue the execution of the caller while the IADT performs its operation, and
- Dynamic reconfiguration:
 - The ability of the IADT to perform bookkeeping operations off-line, with minimal impact to the client programs. In this section we describe the mechanisms of Ada that we used to achieve the above advantages.

Increased parallelism

The ability to have increased parallelism is inherent in the scheme we have established. All interaction with an IADT is conducted via rendezvous with a task we call the task manager. Upon accepting a rendezvous, the manager verifies the precondition of the requested operation. If the precondition is met, the client is released to continue execution, while the task performs the required operation in parallel. If another client request occurs before the previous request is finished, the tasking semantics ensure that the second request is blocked, pending completion of all prior requests. This is because each client request is completely processed before another rendezvous becomes possible. (Section **Future Work** of this paper suggests how this mode can be extended for increased parallelism.) The tasking implementation is hidden from the client program,

requiring no change in client code using a conventional ADT. The only restriction is that all operations must assume the form of procedure calls, a minor restriction considering the potential advantages.

Dynamic reconfiguration

The primary intelligence in IADT's is found in their ability to adjust their representation or behavior according to their run-time history. Adjustments may be desirable to improve space utilization or to decrease execution time. Programs frequently proceed through phases, in which different representations would be better. For example, while creating a database of student records it may be advantageous to use a linked representation to minimize the amount of time necessary for insertions and deletions. Once the student population has stabilized and retrievals dominate the executed operations, it may be advantageous to switch to a sorted array representation to save time and space.

Since our IADT's are implemented via tasking, they can handily accomplish the above scenarios. Each task body consists of a loop that first handles accepts and then performs necessary bookkeeping. The bookkeeping can include actions such as determining whether or not to switch representations or whether or not to perform different cleanup activities. In the text below we discuss the style and intent of the "intelligence" that can be incorporated into IADT's. By factoring such actions to a central location we improve the maintainability of our modules.

It should be noted that the principle of information hiding dictates clear division of labor between client and IADT when it comes to improving efficiency. Information hiding prevents the IADT from making assumptions about the client and the client module from making assumptions about the IADT, beyond that which is documented in the interface. In particular the client should not try to optimize

its performance based upon speculations of how the module is implemented. Neither should the IADT presume to know the space/time trade-off desired by the client. For example, it would be counterproductive for an IADT to optimize for space, when time is crucial.

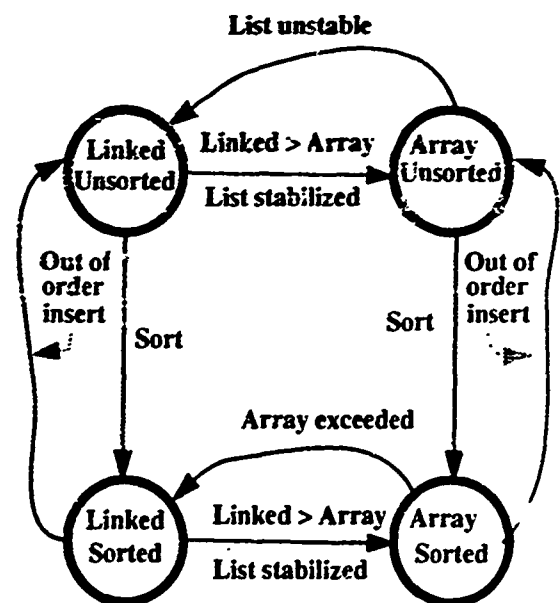
Information hiding encourages the use of assumptions over presumptions. It is therefore important to move presumptions about efficiency into the interface and document them properly, turning them into assumptions. We have therefore included as part of our interface an operation for the client to indicate the kinds of optimization that would be desirable. A call to `List_Efficiency (Desire)` will allow the client to specify `Desire` to be `SpaceOverTime` or `TimeOverSpace`. Processing this request will cause the module to optimize in the desired direction if possible.

How such optimization occurs is up to the IADT. In some situations the IADT can decide for itself on a particular matter of efficiency. For example suppose two representations of lists could be used, a linked (unbounded) representation and a array-based (bounded) representation. Suppose we wish to optimize for space. A linked representation is more efficient up to some threshold, beyond which the array-based implementation is more efficient. After crossing the threshold it is reasonable to switch representations. Similarly, if for a specified period of time the interaction has stabilized in such a way that a linked representation would be more efficient in time (say, when there are many insert/delete operations) it is perhaps appropriate to switch a linked implementation to an array-based implementation.

The key to the intelligence lies in the analysis of when a change of representation or algorithm is necessary. In our preliminary version of IADT's we have specified these changes as finite state machines in which the states denote a particular configuration of algorithms and data structures, and a transition is effected by

the occurrence of certain events. For our list example an event is exceeding a node threshold, exceeding the size of the bounded representation, and producing a particular mix of operations. Our starting state is one in which the list is stored using a pointer. When we reach a state in which the size of the list exceeds the size of array by 10 percent, we switch representation to an array. When the array is filled, we switch back to the linked representation. We keep track of whether or not the list is sorted, updating this information at every insert. As long as it is sorted, we use the more efficient algorithms for searching the list in each representation, switching to linear search when the list becomes unsorted. Finally if `TimeOverSpace` is specified, we switch to the array implementation as soon as the proportion of inserts and deletes falls below 10%.

FIGURE 3. Efficiency Transition Diagram



Future work

The key to maximizing concurrency is to allow the concurrent execution of IADT operation if at all possible. In our initial version operations for a single IADT must be run sequentially

since a second request to an IADT operation will be blocked until the prior one finishes. Such a blockage is not always required, however. For instance, in our list example, it is perfectly reasonable to allow certain combinations of operations to proceed simultaneously (e.g. `Get_List` and `List_Length`).

To solve this problem within the constraints of information hiding, we propose to implement some of the operations themselves as tasks and to augment the preconditions of each operation to check for the validity of the call under the constraints of the currently executing operation tasks. If this augmented precondition is satisfied, then the proposed operation can be executed concurrently with the other executing operations. Violation of the precondition means that processing is still progressing whose completion is essential before the requested operation can be performed.

These blockage preconditions are frequently dependent upon the particular implementation strategy, hence they can be dependent upon lower-level IADT's. The parent IADT can then determine its actions based upon the status of the lower level IADT's.

It is thus necessary not only for an IADT to determine its ability to progress, but also to enable any caller to determine its current ability.

One proposal then is to require every IADT designer to predetermine what operations can execute concurrently, and to provide a public boolean variable (for each operation) to indicate whether or not the operation is currently executable. Information hiding is preserved because how this determination is made is private to the IADT; the fact that it has been made is public. By inspecting these status variables of lower-level IADT's, a higher-level IADT can determine its ability to progress, and queue a request for service or delay the client, depending on the nature of the operation. For example, a sequence of inserts, moves, and deletes can be queued for a list without delaying the client, but

a subsequent get might have to be delayed.

The above discipline assumes there is but one client accessing the list. In the case of multiple clients, the situation is considerably more complicated by the IADT concurrency. Client implementors should not have to worry about concurrency of IADT's, focussing otherwise on ensuring that their sequences of calls to the IADT are issued in the required order. Thus, it is necessary for the IADT to ensure that sequence of issuance is the same as the execution sequence. To do so it may be necessary to use a priority queue with priority determined by the time stamp of the operation request.

Conclusions

We have introduced the concepts of intelligent abstract data types. An IADT offers the potential for increased concurrency and improved space/time trade-off management. We have discussed the features of Ada used to implement an initial version of this concept to provide limited concurrency and dynamic reconfiguration of the underlying representation. Possible extensions of this concept for improved concurrency and efficiency have been proposed.

A REUSABLE Ada MODEL FOR INTERPROCESS COMMUNICATION

Michael J. O'Connor

Teledyne Brown Engineering, Huntsville, Alabama

James W. Hooper

Marshall University, Huntington, West Virginia

and

University of Alabama in Huntsville, Huntsville, Alabama

Abstract

The purpose of this work was to develop an interprocess communication model that could be used in many types of software development projects. The interprocess Message Model is the result. By providing a high level abstraction for a function that is generally both complicated and machine dependent, the IPMM frees the application developer to concentrate on solving his problem

Background

This paper discusses the Interprocess Message Model (IPMM) which is designed to support communications between distributed processes. The IPMM constitutes a versatile process interface, based on Ada, embracing modern software engineering practices. The IPMM is not tied to a specific computer architecture or operating system.

To begin with, what is a process? In the context of this paper, a process is a logically related set of executable code that performs a high-level function in a computer system. For purposes of discussion, processes will be treated as though they were independent tasks at the operating system level. The following

assumptions are made about the behavior of processes:

- Processes are independent of each other; that is, one process can not directly affect the state of another process.
- Processes are scheduled independently and the scheduling is not controlled by the processes.
- Each process has its own memory space.

Many large computer software systems are divided into multiple processes to make the solution to the problem more understandable and to support parallelism. For multiple processes to work together, they must be able to share information. The concept of information sharing is fundamental to all computer systems. Probably the simplest way for processes to share information is for one process to write the data to a file to be read by other processes. While this solution is simple, it is very slow. Database systems are used to allow many processes to access and store information in a central repository. The problem of processes sharing information is much too large to be covered in any one discussion. The problem discussed here is limited to the needs of real-time and near real-time systems to pass messages among multiple processes.

Traditionally, the problem of process communication in a distributed system has been solved on a case-by-case basis. This is particularly true of real-time systems which are required to respond properly to "events" as they occur (where an event is the arrival of a unit of information to a process, thereby affecting the state of the real-time system). Because of the need for rapid processing, other concerns such as flexibility to change have often been neglected. Real-time processing is generally required in military systems, air traffic control systems, nuclear power generation systems, and other time-critical processes. Because of the similarity of near real-time systems to real-time systems, similar methods have been used to implement them. Event processing in near real-time systems is similar to that of real-time systems; however the timing is not as critical. Traditional implementations of near real-time systems were designed for specific hardware and operating systems and are therefore not applicable to the general problem.

Approach

The purpose of the IPMM is to provide an interface that is general enough to be used for a wide range of systems without limiting the implementation to a single hardware/operating system set. The IPMM does not make any assumptions about the operating system or hardware it runs on. While the actual implementation will require the use of system dependent functions, such functions are not reflected in the model interface. The IPMM thus presents a consistent view and behavior as seen by applications using the model.

The goal of the Interprocess Message Model (IPMM) is to simplify the application developer's job. To the application developer, the interface is the most important part of the IPMM because this interface is the only part that he or she

sees. By restricting the developer to the interface only, the abstraction of the communications model is the only part visible. This abstraction provides only the level of detail that the developer needs. The IPMM was designed using the concepts of Object-Oriented Design [COA91].

The IPMM provides the ability to make processes independent of each other. Designing a system with independent processes greatly simplifies the developer's job by allowing the processes to be developed independently. The developer's primary concern is the information received by the process. The processes that use the IPMM should be event driven, where an event is represented by the receipt of a message. This concept of an event should not be confused with an event in a discrete event simulator, in which events are actions which are scheduled to occur at a specific time. With the IPMM, messages are produced when a sending process needs to pass information to a receiving process. In real-time systems, the order of events may not be known beforehand. Event driven processes must be designed to process an event and then to wait for the next event to arrive. While there are several possible ways to implement an event driven process, the Main Event Loop is the approach used by the IPMM.

Implementation

The Main Event Loop is the template for all processes using the IPMM. Ideas of the Main Event Loop in the IPMM were drawn from several sources. One of the best examples of the Main Event Loop can be found in *Inside Macintosh, Volume I*, by Caroline Rose [ROS85]. This text is the guide that is used by developers of applications for the Apple Macintosh. The model used by X-window process is also similar [SMI91].

The Main Event Loop shields the process from changes in other processes. As long as the data the process receives and the actions on the data do not change, modifications in other processes do not affect it. Changing the timing or the order of the messages would not necessarily change the process.

The actual template for the IPMM Main Event Loop is shown in Figure 1. The template, through its use of the Main Event Loop, provides the interface for the IPMM. This interface is a very clean one. An integrated circuit is a very good concrete example of a clean interface to a set of services. A designer is only concerned about what outputs will be generated by a given set of inputs. The exact transformation of the input to the output is hidden. This is analogous to information hiding in software. The IPMM provides a clean interface on the input side by only responding to a defined set of messages. The set of messages that the process will respond to is visible in the Main Event Loop. The Message Identifiers (Ids) that are accepted are listed on the "WHEN" clauses of the case statement. The action taken in response to a message should be contained in a procedure that is defined elsewhere. By removing the details of the action from the Main Event Loop, the overall processing becomes more evident. If the convention is that all MSG_PUTs are also done at the level of the Main Event Loop, then the output of the process is also easily determined.

The process template establishes a pattern for processes using the IPMM. It is also important to have a standard template for defining messages. All messages in a system using the IPMM are visible to at least two independent processes. Each process could conceivably define its own copy of the message. This, however, can lead to several problems. The first is that each process could declare the record, to be used as the message, differently. For

the message passing to work correctly, each process must use the same definition. A second problem is that by defining the message record in two or more processes, the message loses its identity as an object. Therefore it is desirable to declare the message record only once. This can be done by placing the record definition in a package that is "WITHed" by both processes. This can be referred to as the message definition package. Figure 1 shows "WITHing" of a message definition package named MSGDEF.

```
with MESSAGEPKG;
use MESSAGEPKG;
with TEXT_IO; use TEXT_IO;
with MSGDEF;
use MSGDEF;
procedure PROCESS_NAME is
-- An instance of the message
-- defined in the
-- Message_Definition_Package
MESSAGE :
    Message_Record_Definition;
MSG_ID : Message_Id;
begin
loop
MSG_ID := MSG_WAIT;
case MSG_ID is
when Message_Identifier =>
    MSG_GET ( MESSAGE );
    HANDLE_MESSAGE ( MESSAGE );
-- Add when statements for
-- each message to be
-- handled.
when Exit_Message =>
    exit;
when others =>
    PUT_LINE ( "Unknown"
        &"message received" );
end case;
end loop;

end PROCESS_NAME;
```

Figure 1 Process Template

The message object is actually composed of a triple, consisting of the message record, the message identifier, and the instantiation of the generic message package. These components

have been individually discussed in the previous sections. However, it is only when they are combined that a message is fully defined. The three components of the message are encapsulated into a message definition package. Figure 2 provides an example of a message definition package. The "renames" statements are used to make the MSG_PUT and MSG_GET routines directly visible to the process template.

```
with MESSAGEPKG; use MESSAGEPKG;
package STATE_VECTOR_MSG is

  STATE_VECTOR_MSG_ID :
    constant Integer := 1;

  type State_Vector_Msg_Rec is
    record
      VELOCITY : array (1..3) of
        Float;
      ACCELERATION : array (1..3)
        of Float;
      TIME_OF_VALIDITY : Float;
    end record;

  package STATE_VECTOR_MSG_IO
    is new MSG_IO
      ( State_Vector_Msg_Rec );

  MSG_PUT ( Message_Id,
    State_Vector_Msg_Rec ) renames
    STATE_VECTOR_MSG_IO.MSG_PUT

  MSG_GET ( State_Vector_Msg_Rec )
    renames
    STATE_VECTOR_MSG_IO.MSG_GET;

end STATE_VECTOR_MSG;
```

Figure 2 Message Definition Package

A very important feature of the IPMM is how a process determines the arrival of data to be processed. In many systems, a process polls a flag or data store to determine if new data is ready. This reduces the flexibility of the processes and increases the coupling between processes. Polling can also waste valuable CPU time. In the IPMM, a process

does not poll, but rather "sleeps" until a message arrives.

The Interprocess Message Model (IPMM) concept is independent of both the host hardware and the host operating system, as has been discussed in the previous sections. This independence is possible because Ada allows most of the actual implementation of the model to be achieved without regard to the hardware or operating system. Only a small part of the actual code is dependent on the target environment, and this small part is hidden from the user of the model's services. Detailed discusses the IPMM's implementation in for both VMS and Unix are provided in [OCO91].

Conclusions

The purpose of this work was to develop a model that could be used in many types of software development projects. The Interprocess Message Model is the result. By providing a high level abstraction for a function that is generally both complicated and machine dependent, the IPMM frees the application developer to concentrate on solving his problem. Two primary conclusions can be drawn from this work: (a) the IPMM can be used to solve the general problem of interprocess communication; and (b) it is possible to create a high-level abstraction that can be reused.

Mary Shaw states that "Engineering relies on codifying scientific knowledge about a technological problem domain in a form that is directly useful to the practitioner, thereby providing answers for questions that commonly occur in practice" [SHW90]. The idea is that a solution to a common problem can be applied over and over to the same type of problem. The IPMM is a general solution to the problem of interprocess communication in multi-process systems. While the IPMM cannot

be used in all multi-process systems, it can be used in many.

When software developers have been presented with the IPMM as a basis for their software development activity, they have been generally resistive. Developers can generally find many reasons for not using the IPMM, with the most common being: "it will be too slow", "it is too difficult to understand", "it will not work", and the ever present "I don't like it." The IPMM is subject to the same problem as all reusable components, the "Not Invented Here" syndrome. A model similar to the IPMM is essential to the development of large real-time systems. However, given their choice, many software developers would prefer an ad hoc solution to interprocess communication.

Some developers find using the Main Event Loop distasteful. They resent having to structure their code by a pattern supplied by someone else. This is the view that software development is based on individual creativity or artistic ability. However, most developers have felt positively about the IPMM after using it. The primary reason for the change in opinion has been that the developers realized the model was useful, and that it allowed them to concentrate on solving their problem.

The reuse of software components is discussed at great length in the current literature. *Software Reuse : Guidelines and Methods* by J. W. Hooper and R. O. Chester discusses the current state of software reuse [HOO91]. Reuse of components on the level of Booch's Ada components prevents commonly-used routines from needing to be rewritten by every developer on a project [BOO87]. Biggerstaff and Perlis state that the larger the reusable component the larger the savings to a project that uses it [BIG89]. However, they caution that large components are much more difficult to reuse. Because the IPMM is a large

component that provides a clear abstraction of the underlying problem, the benefit to a project using it should be great. The general nature of both the problem and the solution makes the IPMM applicable to a large number of systems.

The use of the IPMM should be considered when a system is designed and implemented. To use the IPMM, the developer must follow the IPMM model in design and implementation. In other engineering disciplines, solving a problem by using a standard model is the norm. An engineer attempting to apply a non-standard solution to a common problem would have to provide extensive justification and proof that the method would work. A civil engineer charged with building a bridge over a small river would consult existing books on the subject and select an appropriate approach. The engineer would not have to resolve the problems of statics or strengths of materials. This is an example of Shaw's concept of an ordinary practitioner applying existing knowledge to common problem [SHW90]. The civil engineer would not feel that his creative abilities were being stifled by using a standard model. It is unfortunate that software developers often feel constrained when using standard models. Large-scale reuse will only be possible when many models based on high levels of abstraction are developed.

Bibliography

- [BIG89] Biggerstaff, Ted J. and Alan J. Perlis, *Software Reusability Volume 1 Concepts and Models*, ACM Press Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.
- [BOO87] Booch, Grady, *Software Components with Ada*, The Benjamin/Cummings Publishing

Company, Inc., Menlo Park, California, 1987.

[COA91] Coad, Peter and Edward Yourdon, *Object-Oriented Design*, Yourdon Press, Englewood Cliffs, New Jersey, 1991.

[HOO91] Hooper, James W. and Rowena O. Chester, *Software Reuse: Guidelines and Methods*, Plenum Press, New York, 1990.

[OCO91] O'Connor, Michael J., "An Ada Based Model for Interprocess Communication", M.S. Thesis, The University of Alabama in Huntsville, 1991.

[ROS85] Rose, Caroline, *Inside Macintosh, Volume I*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1985.

[SHW90] Shaw, Mary, "Prospects for an Engineering Discipline of Software," *IEEE Software*, Vol 7, November 1990.

[SMI91] Smith, Jerry D., *Object-Oriented Programming with the X Window System ToolKits*, John Wiley & Sons, Inc., New York, 1991.

Biographies

MICHAEL J. O'Connor is currently employed as an engineer for Teledyne Brown Engineering in Huntsville, Alabama. He serves as technical lead for applications on a large software development effort. He holds a Bachelors of Computer Engineering from Auburn University and a M.S. in Computer Science from The University of Alabama in Huntsville. He specializes in the development of large distributed real-time systems in Ada for defense applications.

JAMES W. HOOPER currently serves as Weisberg Professor of Software Engineering at Marshall University Huntington, West Virginia, on leave-of-absence from the position of Professor of Computer Science at The University of Alabama in Huntsville (UAH). He emphasizes software engineering and programming languages in teaching and research. He holds B.S. and M.S. degrees in Mathematics, and M.S. and Ph.D. degrees in Computer Science. Prior to joining UAH in 1980, he was employed by NASA Marshall Space Flight Center, where he conducted research in simulation approaches for NASA missions.

USAISSDCL In The Trenches With Ada

LTC PAUL D. BATES
US ARMY

BART JEFFCOAT
COMPUTER SCIENCES CORPORATION

ABSTRACT:

The United States Army Information Systems Software Development Center-Lee (USAISSDCL) is responsible for developing management information systems that support automated Army logistics requirements. In an effort to reduce software development and maintenance costs, USAISSDCL has investigated the applicability of Ada and Ada/SQL binding modules for logistics management information systems (LMIS). This paper focuses on the lessons learned during the redesign of the Standard Army Ammunition System Level 1/3 (SAAS1/3). Lessons learned are presented in the areas of training, team composition, Ada Programming Support Environment (APSE), Ada versus 4GLs, development methodology, reuseability and software architecture. The SAAS1/3 project has been very successful. The use of Ada implemented through a standard architecture in a large MIS development facility can lead to significant reductions in software development schedules.

INTRODUCTION:

The United States Army Software Development Center-Lee (USAISSDCL) located at Fort Lee, Virginia is responsible for designing, developing, fielding and maintaining Standard Army Management Information Systems (STAMIS) that support the United States Army's automated logistics requirements. USAISSDCL is comprised of approximately 760 military, civil service and contractor support personnel. The command is responsible for thirty-two different logistic STAMIS. Four systems are in the concept phase, eleven are in the development phase and seventeen are in the maintenance phase of the life cycle. USAISSDCL has systems operating worldwide at over 1000 locations. The systems operate on fifteen different platforms, and are written in nine different languages. In order to reduce development and software maintenance costs and provide for more flexibility of personnel, USAISSDCL is investigating the applicability of Ada and a standard software architecture using Ada/SQL binding modules that access a relational database for all new development and major redesign projects. The redesign of the Standard Army Ammunition System Level 1/3 (SAAS1/3) was selected to be the target project to determine the applicability of Ada and the Ada/SQL binding module architecture for logistic STAMIS. After a short background the remainder of this paper will focus on our lessons learned from being "in the trenches with Ada."

BACKGROUND:

SAAS1/3 was originally fielded in 1973. It is the second and third tier of a three tier system that provides visibility to National Inventory Control Points of an estimated fifty billion dollars of conventional ammunition. Visibility is maintained by condition code, serial number, lot number, component and quantity. It is currently operating at 13 locations worldwide. The procurement of a new hardware platform and significant

enhancements to user requirements dictated the total redesign of SAAS-1/3. The project is scheduled for fielding in July 1992. The hardware platform for the SAAS1/3 redesign project is the Unisys 5000/95 minicomputer referred to as a Corps Theater Automated Service Center (CTASC-II). The CTASC-II is a 32 bit machine with 2 central processing units, fourteen 1.02 GB hard drives, and 65 MB of system memory. The operating system is UNIX 5.3. The fielded system will utilize Unisys Desk Top III microcomputers as terminals. Software development is being completed in Ada employing a commercial relational database product, Extended Database (XDB). The database is accessed through Ada/SQL binding modules that are ANSI compliant and are compiled as any other Ada program.

TRAINING:

One of the first major decisions concerned the type, quality and quantity of training required in Ada, relational database design and the UNIX operating system. Although the SAAS1/3 team was well motivated and very experienced in COBOL, the team had little experience in designing relational database systems, implementing the Ada/SQL binding modules and programming in Ada. With the support and encouragement of management the following training program was established for the entire SAAS-1/3 Team:

COURSE	DURATION
Database Design/SQL	1wk
Ada Concepts	1wk
Ada Programming/methodologies	3wks
Advance Ada Programming	4wks
Unix	1wk

In addition, mini in-depth training classes were established throughout the project to address specific technical issues with Ada, XDB and UNIX. The training program was intentionally designed to teach the skills necessary to take advantage of the entire Ada environment, not just the features of the new language. In order to prevent student burnout, training was conducted over a nine month period. Everyone (management and team members) held the perception that an extensive Ada training program would produce Ada experts. However, this proved not to be the case. In most cases a strong COBOL programmer became a strong Ada programmer and a weak COBOL programmer became a weak Ada programmer. There were some instances where good COBOL programmers had a difficult time accepting the structure and modularity of Ada. Due to delays in the delivery of hardware and software there was a three to twelve month time lag between Ada training and actual system design and development. This caused a noticeable impact on productivity. Ada is like any other skill, if you don't use it, you lose it.

Lessons Learned (Training):

- Ada training is much more than just learning a new programming language
- everyone doesn't become an Ada expert by going

through an extensive training program

- the benefits of Ada training will be lost if not immediately utilized after training
- expect a six month learning curve before most programmers become competent in using Ada

ADA PROGRAMMING SUPPORT ENVIRONMENT (APSE):

The APSE initially consisted of an Ada compiler and a very limited reusable component library developed with in-house personnel. This was primarily because the CTASC-II is a non industry standard. Commercial off the shelf software was difficult to find. A limited APSE consisting of a debugger, formatter, optimizer and profiler was pieced together nine months into the project. Resource limitations drove the decision to develop an automated library tool. Hind sight tells us it would have been better to procure a commercial product. Nine months of senior programmer effort was spent developing our automated library tool. Commercial products generally sell in the \$35-\$75k range. The requirement to have a robust automated library tool can not be overstated. According to the STEELMAN requirements (1), there should be support for large scale development in the form of libraries that contain generic definitions and separately translated units. This equates to separate compilation and linking capability and provides Ada development with the ability to build executable images from those units. With this type of support along with a configuration management philosophy, it is easy to develop and maintain library versions without harming the project's productivity. Medium to large scale Ada projects must have automated library support. A manager with a configuration management tool can keep projects from committing developmental suicide and save the project considerable time by eliminating the need to reduplicate work. During the development of SAAS1/3, numerous upgrades to the operating system and database had to be assimilated. Without an automated library tool and personnel who thoroughly understand how to employ it, the SAAS1/3 project would not have maintained established schedules. For example, an upgrade to the Ada-SQL binding interface may not compile with previously compiled and executing versions. In this situation the project cannot stop until a resolution is found and the switch must be made back to the working version. An automated library tool makes this switch possible. Library support takes on additional significance when a project is in a maintenance phase. Any slip in the configuration management schema could cause the entire system's destruction. It is precisely this reason configuration management must be understood completely by management and personnel in control of this tool. The other side is that configuration management competently exercised can and will provide smooth and almost painless transitions to modification of project affected code.

Lessons Learned (APSE):

- Ada development requires more than a compiler. As a minimum, a debugger, optimizer, access to reusable libraries and an automated library tool must be on hand
- the librarian requires an in-depth understanding of Ada concepts
- although the cost seems high, procuring an established, well designed automated library tool is well worth the effort

SOFTWARE ARCHITECTURE:

The objectives for the SAAS1/3 architecture were simple. We wanted an architecture that would:

- limit commitment to vendor specific software
- port to all STAMIS
- be understandable and useable by all programmers

- apply to the entire range of project requirements (data access, communications and technical requirements)
- support reuse

The architecture designed to reach these objectives is simple but robust. It is illustrated in chart 1. The database is used to store application and reusable package data. For example, the user interface required screen titles, form control numbers and classifications, report generation expressions, headings, classifications and report control numbers, and job control processing expected process control numbers, parameter and transaction file names, execution status flags, etc. All of these are contained on tables within the database. In addition, all interaction with the database is accomplished through the module compiler. SAAS1/3 decided to use the Ada-SQL module compiler to produce the binding interface exclusively. This architecture enforced standardization of the database interface and quickly trained staff members how to develop interfaces to a relational database. Also, by using this methodology, we were using a compiler that closely followed ANSI SQL standards and met the recommended SQL-Ada Module Extension (SAME) methodology. SAAS1/3 kept the module compiler produced Ada packages separate thereby reducing impacts due to changes in binding. This architecture reduces the risk and labor involved in changing database vendors. SAAS1/3 has the freedom to switch to other vendor database products that support the SAME recommendations with a minimum of effort.

Much has been written about the pros and cons of using a 4GL versus Ada. Current evidence suggests that Ada is more efficient for medium to large size systems (2). Ada has many benefits over 4GLs for STAMIS systems. From the SAAS1/3 perspective, Ada when implemented utilizing a mature reusable component library out performs 4GLs. Once our reuse library was completed, programmers could put together modules as quickly in Ada as with any 4GL. In addition, Ada is nonproprietary. We found it easier to teach programmers how to combine Ada packages out of the reuse library than to teach the specifics of a 4GL. Twelve months into the project four additional programmers were assigned. These were Pascal programmers with no Ada training. These programmers became productive after only a few days of orientation.

Lessons Learned (Software Architecture):

- Ada systems do not have to be built around proprietary software
- Ada out performs 4GLs when used with a mature reuse library

TEAM COMPOSITION:

Prior to the start of the redesign, the SAAS1/3 team was organized around a team leader who supervised five programmers. The assistance of a technical writer/editor was available on an as needed basis. The team had worked together for many years and had developed numerous unwritten procedures that defined work procedures and identified the portion of the system each person was responsible for maintaining. Each team member knew how to test their part of the system with well established bench mark test files. The Team Leader was the technical expert on the team. Librarian duties were performed as an extra duty by one programmer. The team had an excellent reputation for delivering timely quality software change packages. In preparation for the redesign effort, the decision was made to increase the size of the team to nineteen including the team leader. This decision was made in order to schedule all training at one time to become proficient in the Ada language before being subjected to the pressures of development time schedules. Although user requirements were not completely defined for the

redesign project, work year estimates were determined based upon original functionality. Five of the additional thirteen team members were contractors. Although the contractor staff had their own supervisor that reported to the team leader for assignments, the contractors participated in all team planning. The original team concept for the redesign project was to have a core group of programmers that supported a lead designer that worked in conjunction with the team leader on the technical aspects of the project. Each programmer would still be responsible for unit level testing. Early in the program it became clear this concept was inadequate. Over the period of several months, team roles changed substantially to accommodate the scope of the redesign project and the requirements of structured design methodology implementing Ada. Change did not come easily and the team experienced traditional growing pains. The current Team Configuration and roles are:

Team Leader: no longer the technical expert, establishes/monitors schedules, plans future actions, coordinates with users and upper management

Lead Designer: Ada technician, enforces standards, conducts reviews

Librarian: monitors compliance with standards, maintains and/or develops Configuration Management (CM) tools

Database Administrator: establishes and maintains data dictionary/schema, maintains the different databases necessary for development

Tester: coordinates and conducts incremental testing, maintains problem report logs

Trainer: full-time dedicated trainer (Ada proficient), provides technical assistance to team members on an as needed basis, allows the lead designer to focus his energies on design and global issues

Systems Administrator: maintains system performance and other traditional duties

Communication/Security Sub Team: designs and develops the data security and data communication aspects of the system

Lessons Learned (Team Composition):

- Ada development requires a well structured team with clearly defined roles
- the identification of a team trainer increases productivity providing assistance on an as needed basis
- team roles will change as the project matures, plan for it

REUSEABILITY:

"A carefully engineered collection of reusable software components can reduce the cost of software development, improve the quality of software products, and accelerate software production"(3). The search for reusable software components is a vital part of any successful Ada project. SAAS1/3 found numerous reusable packages from vendors, the Ada Software Repository, the Reusable Ada Products for Information Systems Development Center, in addition to developing several sophisticated packages in house. In order to increase standardization, reduce repetitious coding and decrease complexity, experienced programmers were tasked to develop reusable interface packages that encapsulated several software components. An excellent example of this technique is the screen input/output interface modules developed by the team. The interface modules enforced standardization of windows, menus, forms, messages, scrolling and selection

techniques, error handling, etc. Development of the reusable interface packages contributed greatly to a reduction of development time, enforced adherence to standards, and provided explicit examples of sound software engineering principles. Most projects, SAAS1/3 included, are required to implement numerous Department of the Army standards that have been captured in text documents. These standards must be interpreted by the programmer during development. Examples of these types of standards are screen layouts and report formats. Ada's strong structure and portability allow standards to be implemented through reusable packages. This integration of standards into reusable packages prevent independent interpretation. In addition, changes to standards can be implemented by merely changing a database table or recompiling and linking the reuse library. In the future, standards must be integrated into reusable packages and not be left up to interpretation.

Lessons Learned (Reuseability):

- designing for reusability is the key to a successful Ada development
- reuse takes all the guess work out of interpreting standards; the standards are incorporated into the reuse modules
- time spent searching for reusable components pays big dividends

DEVELOPMENT METHODOLOGY:

SAAS1/3 did not use an automated integrated development methodology such as those provided with several vendor CASE tools. As a result, a software engineering guide had to be developed. A modified object oriented development methodology was created by the team as a result of an Employee Involvement Team initiative. The methodology centered around incremental development and incremental testing as shown in chart 2. It is based on sound software engineering practices and guides contained in the NASA Ada Style Guide, MIL-STD 7935, and MIL-STD 2167. The team's methodology required the production of a software engineering notebook (SEN) which replaced the traditional maintenance manual and system specification. As depicted in chart 3, SENs are process oriented and therefore do not require a completed functional specification. A SEN was created for each process in the system. In addition, a developers guide was produced. Every team member actively participated in the development of these two documents and, as a result, felt strong ownership and commitment to see the methodology implemented. The SEN and developers guide were instrumental in instructing the less experienced personnel in acceptable techniques and allowed the more experienced personnel to concentrate on critical tasks. Table A depicts the contents of the developers guide. Table B illustrates the contents of the SAAS1/3 SEN.

DEVELOPERS GUIDE CONTENTS

Analysis and Design steps
Ada Coding Guidelines
Ada-SQL Coding Guidelines
Forms Standards
Menu Standards
Report Standards
Module Naming Standards
Testing Standards
Documentation Standards
Types of Messages and Uses
Processing Methods
Technical Notes

TABLE A

SEN CONTENTS

Functional Definition
Requirements Outline
Component Analysis Checklist
Walkthrough Results
Job Control Interface Classification
Process Impact Estimate
Process Implementation Description
Requirements Implementation Matrix
Input and Output Formats
Test Condition Requirements

TABLE B

Lessons Learned (Development Methodology):

- it is difficult to "find" a methodology that exactly fits your organizations needs. Modify one that supports your organization
- having a development methodology that is documented, understood and supported by everyone on the project is more important than trying to use the methodology currently in vogue
- be creative, Ada supports a wide range of methodologies

DOCUMENTATION:

Process documentation was a key consideration in the SAAS1/3 project. SAAS1/3 does not have an automated documentation tool. However, strict standards were enforced in order to diminish technical program documentation. To the extent possible, the project standards required programmers to take advantage of Ada's inherent documentation features(4) and produce well documented source code. Experience has shown us that in a maintenance environment the first place a programmer turns to is usually the code not the maintenance manual. This type of program development takes a little more time but is well worth the effort. The end results are programs which are self explanatory, standard and understandable. In addition, the ability to conduct thorough reviews and walkthroughs is greatly increased.

Lesson Learned (Documentation):

- efforts spent documenting programs enhance maintenance productivity

CONCLUSION:

Ada has shown it's applicability and effectiveness in the MIS world. Ada can limit dependence on proprietary software, is portable to different platforms, can simplify documentation and enhance standardization. Coupled with a standard architecture, the use of Ada does reduce software development time. Chart 4 shows the SAAS1/3 project broken out into specific areas of system data. Chart 5 shows the interaction between Ada and the SQL binding interface. The next project that uses this software architecture and reusable library should be able to shorten their development schedule by 15-20%. This estimate is based on the time traditionally spent on design and implementation of an application architecture. Proliferation of the SAAS1/3 architecture has been started. The use of Ada at USAISSDCL is a success story. The ultimate success of Ada is up to DOD and its commitment to standardizing software development.

References

- (1) "Requirements for High Order Programming Language, STEELMAN", Department of Defense, Jan 1983.
- (2) "Software Science and Complexity Analysis of Ada and a 4GL", Defense Systems Management College, Feb 1991.
- (3) Booch, G. 1987. Software Components with Ada, preface.

(4) Iohbiah, Barnes, Firth, Woodger, Honeywell, Alsys(1986), Rationale for the Design of the Ada Programming Language.

XDB is a registered trademark of XDB.

UNISYS is a registered trademark of UNISYS.

UNIX is a registered trademark of AT&T Bell Laboratories.

BIBLIOGRAPHY:

LTC Paul D. Bates is the Chief of the Combat Systems Support Division at USAISSDCL, Ft. Lee, Va. He has a Masters of Information Systems from Georgia State University. LTC Bates is an avid supporter of Ada and has been instrumental in developing and implementing software development methodologies that support concurrent engineering and incremental development.

Mailing Address:

LTC Paul D. Bates
14410 Creek Stone Dr.
Chesterfield, VA 23832

LTC Paul D. Bates USAISSDCL
ATTN: ASQB-ILA-C
STOP L-96
Fort Lee, Va 23801-6065
PH# (AU) 687-1425
(COMM) (804) 734-1425

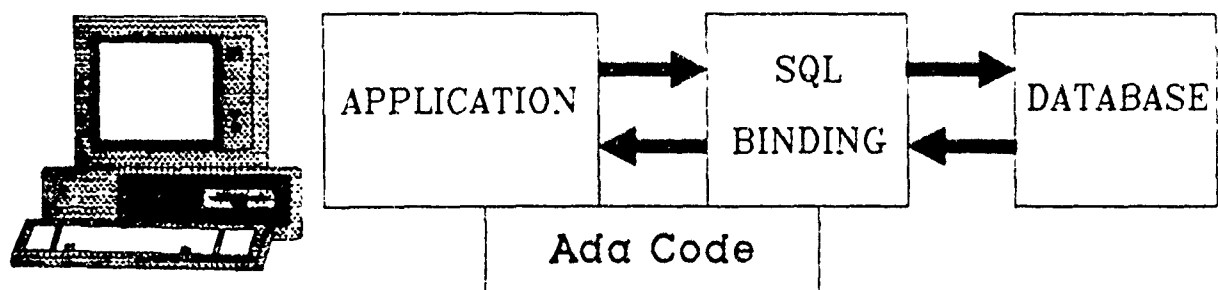
Bart Jeffcoat is a senior member of the technical staff for the Computer Sciences Corporation at Ft. Lee, Va. He has a B.A. from Augusta College and is pursuing a M.S. from Virginia Commonwealth University. Mr. Jeffcoat is a former instructor for the U.S. Army Computer Science School and was a technical team leader on the Standard Army Financial Systems Redesign project responsible for Ada CASE tools including code generators, design and control repositories and library support.

Mailing Address:

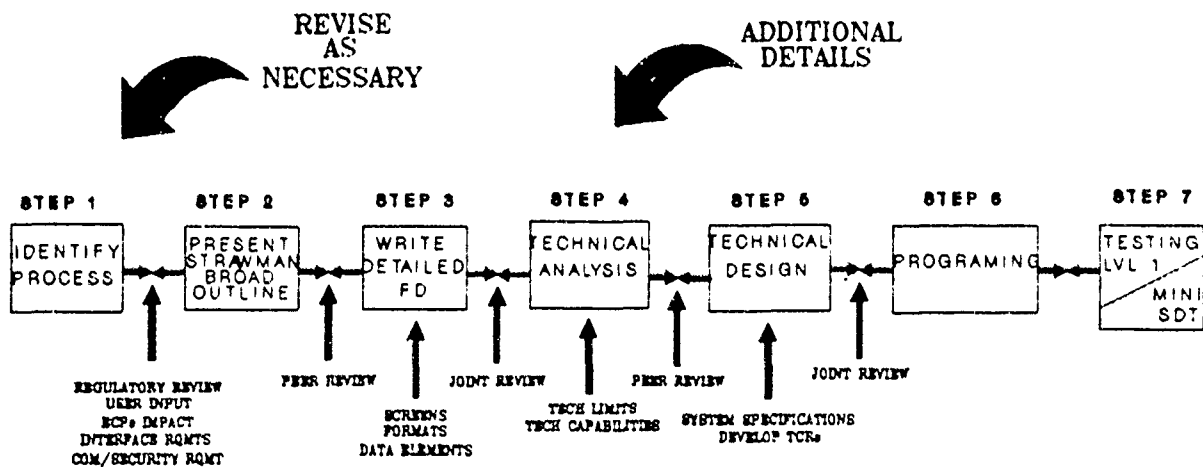
Bart Jeffcoat
P.O. Box 1412
Hopewell, Va. 23860
(COMM) (804) 734-1897

SAAS 1/3

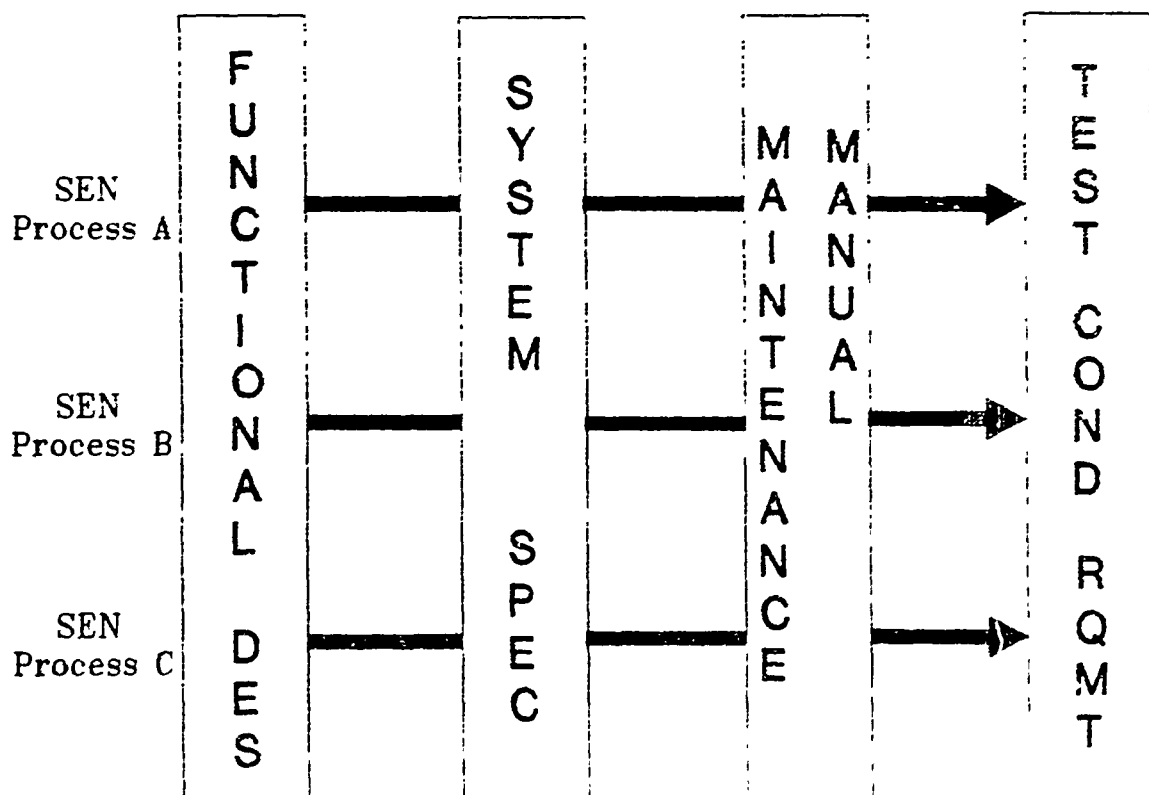
SOFTWARE ARCHITECTURE



SAAS 1/3 PROCESS DEVELOPMENT



SAAS 1/3 INCREMENTAL DOCUMENTATION



SAAS 1/3

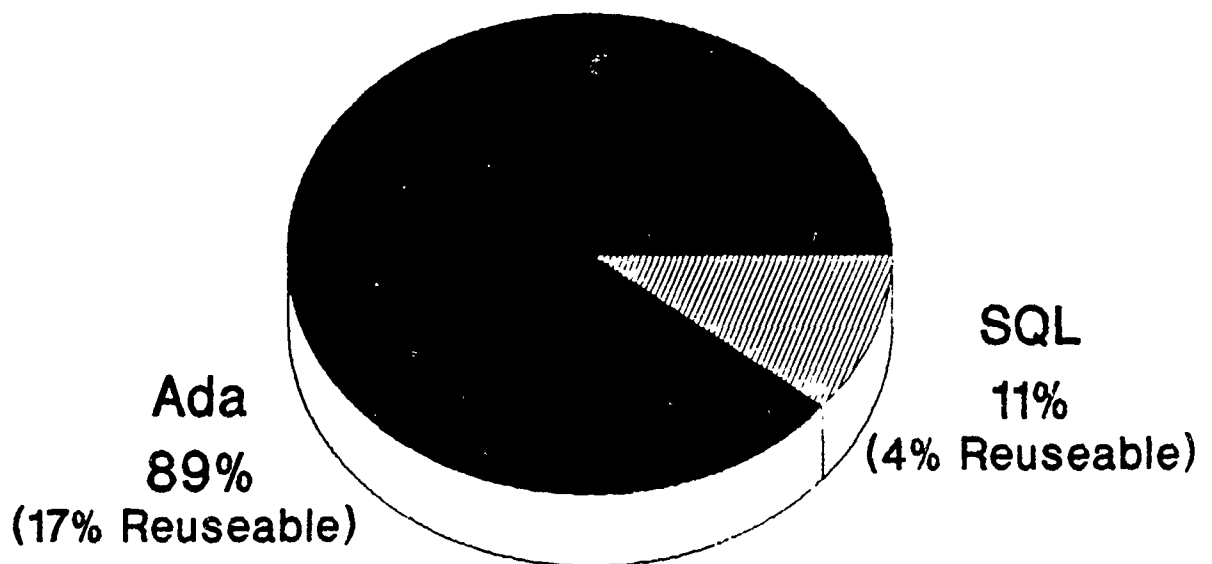
QUANTITATIVE SYSTEM DATA

	Projected
Lines of Ada Code	*140,000
Lines of SQL Code	*14,000
Functional Reuse	**15,000
Lines of Local Reuse	6,000
Programs	45
Local Reuseable Modules	83
Main Database Tables	40

*LOC = NON BLANK/NON COMMENT LINES

**LINES OF FUNCTIONAL REUSE ARE INCLUDED
IN FUNCTIONAL Ada/SQL CODE.

SAAS 1/3 Ada/SQL Percentages



SMALL BUSINESS EFFORTS TO BECOME ADA COMPETITIVE HOW CAN THE LITTLE GUY GET INTO THE GAME?

Joseph P. Hoolihan and Lindon J. Corbett
J.G. Van Dyke and Associates
Annapolis Junction, MD 20701

Summary

The Ada world is rapidly expanding as investors in the Ada vision reap the harvest of their efforts. Some of these early investors have sizable resources and are able to invest without the threat of financial ruin. Newcomers to this arena find both excitement and challenge as they adjust their resources to get on board. Small businesses find this adventure most challenging and almost insurmountable as they discover the uniqueness of using Ada. This paper attempts to define the elements that create this challenge to the small business and present some possible avenues by which the small business may be able to gain a part in the Ada vision without risking resources it does not have.

Ada. It's the LAW! So proclaims a tee-shirt seen at Tri-Ada '90. With the DoD budget of FY 91, this is the case. However, the language itself has been around now for over ten years. There are now 286 compilers available on a wide variety of platforms, and the contracts are out there, waiting for bidders.

But, Wait! Are you ready to bid? Are your people trained? Do you understand the market, what it means to "do Ada"?

A common response to these questions is "Quick! Call in the trainers! Shell out the bucks! Buy hardware, buy software! Hire new people!"

Unfortunately, doing things this way is VERY costly.

Ada was quite expensive in the beginning. Compilers were scarce, and high priced, trainers were few and far between, and Ada experienced people were rare. The major players in this market were of necessity larger companies that had the resources and time to build an Ada organization from the ground up. And, as the big companies won contracts, they were in a better position to win further Ada contracts (since now they had the experience). This cycle continues today. How can a small business compete with these giants, and enter the Ada arena?

One obvious way is to team with a big player as a subcontractor, and gain the experience in that way. However, since most of these giants don't particularly want to spend their profit dollars training someone else's personnel, it becomes imperative for a small business to have some basis to be Ada competitive.

Challenges to Small Businesses

Management Indifference

Many managers have adopted either a "wait and see" attitude towards Ada, or have focused on what brings in revenue NOW, and do not have a good understanding of what the Ada market is all about.

High Cost of Doing Ada

The financial investment can be high. Most compilers are in the several thousand dollar range, and tools and support software can range into the tens of thousands. Ada training is also costly, and there continues to be a shortage of Ada trained personnel.

Lack of a Compatible Engineering Policy

Most companies have become steeped in the waterfall life cycle model of development, and have tailored design and development strategies to fit each new project. Ada lends itself to standardizing development strategies, but managers are reluctant to learn (and try) a different way of building software.

Small Businesses can't Do Ada

There is a general perception that a small business just doesn't have what it takes to do Ada development.

Historical Perspective

To illustrate how a small business can do this without going bankrupt, it will be necessary to digress, and give a personal historical account of how J.G. Van Dyke and Associates went about it.

The authors have been with the company for over 9 years, having joined in 1982. In the Fall of 1983, Mr. Hoolihan attended the Annual ACM conference in New York. At that conference, he heard Jean Sammet give her now famous talk on "How Ada is not Just another Programming Language". He also attended three separate seminars on the Ada language, and signed the petition to make AdaTec a full fledged SIG. From that point on, he became an Ada proponent.

Upon returning from the conference, an enthusiastic trip report was written, full of glowing words on this new and powerful language. It was met with a great deal of indifference. After speaking with our company president (an advantage of being a small business), he stated that "He'd wait and see. After all, COBOL was designed by the Government, and look where it is today."

Somewhat deflated, but undaunted, the authors began to look around at what COULD be done, alone, or with minimal aid from the company. All the companies

that had anything to do with Ada were called, to get on their mailing lists. Numerous publications were scanned for Ada related articles, and a file kept on Ada related articles. As the Ada industry matured, more and more Ada specific advertisements and articles appeared. The authors endeavored to keep abreast of all of them, even asking colleagues to bring such matters to their attention.

In 1984, Mr. Hoolihan attended the Second Annual Conference on Ada Technology, held in Hampton Roads, VA. As with the ACM conference, the company paid salary and attendance fees, and he paid travel and lodging expenses. At this conference, it was made apparent how committed were the proponents of Ada. Also, it was a good introduction to Jean Ichbiah, the father of Ada.

More conferences followed in '85, '87, '88, '89, and '90. At each one, the tutorial day at the start of the conference was attended. First Introduction to Ada, then Advanced Ada, Ada for Managers, Tasking and Generics, Object oriented design, and others.

As the company name became available on various attendee lists (and from SIGAda), Ada information began pouring in. A separate Ada filing cabinet was established, which grew from a few files in 1983, to four full drawers today. Various file topics such as compilers, tools, conferences, newsletters, and CASE, all Ada related, began to grow.

In June of 1986, Mr. Hoolihan purchased the pre-validation release of Meridian Ada, version 1.0. At last, we could do some Ada programming! We also began building up our personal Ada library; some books were purchased by the authors, others were bought by the company. The compiler was upgraded, and in 1989, the company agreed to pay for the annual maintenance fee, thus ensuring continued upgrades.

After the release of DoD directive 3405.1 and .2, upper management began to take a closer look at this "new" language. We began to receive calls from proposal writers, looking for Ada verbiage to add to the effort. Our company has submitted several proposals under the

Small Business Innovative Research program, for various Ada related concerns. Our General Manager has attended the "Ada for Managers" seminar, and we have Technical Writers trained in DoD-STD-2167A.

Today, we still haven't landed an Ada contract, but it figures prominently in a good number of the proposals we write. We continue to attend conferences, and have given "orientation" briefings to management and development personnel, in order to increase awareness.

Also, we have developed a Corporate Software Development Plan, designed to standardize procedures and policies, yet be flexible enough to be tailored to specific projects. This gives us an additional tool not only during design, but is a plus in proposals as well.

Useful Suggestions

So, what suggestions can be gotten from this little tale? Several come to mind.

1. Get on mailing lists. This sometimes happens automatically (if you visited any booth at the exhibits, for instance), or can be done directly. We have found vendors to be more than glad to add your company to their lists. We usually tell vendors what our situation is; we're "information gatherers", not "ready to buy" persons. This has not caused any grief with any vendors.

2. Buy a compiler, and use it. You have to start somewhere, and there are a number of low cost PC compilers available. Begin by doing in Ada the little routines you might do in other languages. Our first Ada program calculated gas mileage. Become familiar with the language. By doing smaller programs, you can ease into the large body (pun intended) of knowledge that is Ada, and be that much closer to fluency.

3. Attend conferences where possible. Our agreement with our company is that they pay salary and conference fees, and we pay all travel and lodging expenses. Thus, local conferences are best, but those farther away can be attended, if your personal budget can stand it. If you cannot attend, then by all means purchase the conference

proceedings, usually available a few months after the conference. Sometimes you can purchase proceedings of previous conferences at a current one. While some information may be dated, a great deal of it remains germane. We have accessed copies back to 1984 to find information on a particular topic.

4. Join ACM SIGAda, and a Local SIG. Not only does this get you on mailing lists, but it is an excellent way to keep in the mainstream of information. Also, the bimonthly SIGAda Letters contains great quantities of Ada information, including specific Ada problems (Doug Bryans Dear Ada column), upcoming seminars, and of course, numerous technical papers. SIGAda also has a number of Working groups, such as the Standards WG (DoD 2167A et al), the Reuse WG, RunTime Environment WG, Performance Issues WG, Object-Oriented WG, and Commercial Ada Users WG. Most of these groups put out periodic newsletters and reports, which provide up to date information on Ada activities.

5. Read. Buy a book on Ada, and read it, cover to cover. You won't catch all the information the first time, but you will get exposed to an overview of Ada. Also, there are numerous free publications that contain Ada related information. Sign up and READ. Get your associates to bring Ada articles to your attention.

6. Establish an Ada information repository for the company. When Van Dyke and Associates first became interested, the repository was a small stack of compiler companies' information brochures in one corner of one bookshelf. Today, we have managed to fill a large, four-drawer file cabinet, and start on a second one. Company personnel can access and obtain information for use in proposals and client presentations.

7. Take advantage of any training opportunities that become available. While most commercial training seminars are quite costly (up to \$800 per person per day of training), there are frequently lower priced training opportunities. The tutorials given at the beginning of this conference are a good example. Local SIGs frequently

have one day sessions, as do the local ACM and IEEE chapters.

8. Become an Affiliate of the Software Engineering Institute. This is a program run by SEI at Carnegie Mellon University to facilitate the dissemination of information of new technologies and methods. Various publications are available from the SEI to affiliate members. Seminars and Symposia are also available; members are kept informed of schedules.

9. Develop a Corporate Software Development Plan. DoD 2167A gives instructions on format and content. Formalize the information that will be consistent from one project to the next, and delineate which sections are to be customized on a per project basis. This not only puts you ahead during design, but will help the company focus its attention on the process. Familiarity with DoD 2167A, 2168, and Mil Handbook 287 (2167A Tailoring Guide) will also be useful in both proposal writing and in being better prepared to begin development efforts.

10. Talk to your Management about Ada. Repeatedly. Once you are well versed in what Ada is about, you can converse intelligently with Management as to its features, special considerations, and points of interest unique to Ada. This process can be lengthy, and sometimes frustrating. You may feel like the "lone voice crying in the wilderness", but persevere. At the very least, you are making yourself more knowledgeable in Ada, and at best, you will become a valued asset when the company is finally hit with the reality that "It's the LAW!"

Business Opportunities for Small Businesses

Small Business Innovative Research(SBIR) Program

The definition of a small business is not more than 500 employees, and annual revenue not in excess of \$12.5 million. This definition comes from several articles in the Code of Federal Regulations, and is used by various Government Agencies to determine eligibility for Small Business programs. The primary one of interest to those in the Ada community is the Small Business Innovative Research program, or SBIR. Several times a

year the DoD issues a Document, listing RFPs for small businesses. These cover the various DoD components, as well as NASA, FAA, and others. The program is designed to operate in three levels. The first is a flat \$50,000, six-month effort, usually to prove the concept. The second level is a \$250,000, six-month effort, to attain a certain level of capability from the concept. The third level entails full proposal efforts, with a full blown development contract, and major funding. Each level depends on the previous one, and a company must complete each level in order to bid on the next.

Ada topics in the past have ranged from feasibility studies, to compiler evaluation, code conversion, and embedded applications. Many of the proposals listed in this document fail to get funded; the sponsoring agency may not get any bidders, or the bidders may all fall short in the sponsoring agency's opinion. Still, it is an excellent way for a small business to start on the Ada road. To obtain a copy of the SBIR solicitation contact:

Mr. Bob Wrenn
SBIR Coordinator
OSD/SADBU
U.S. Department of Defense
The Pentagon - Room 2A340
Washington, DC 20301-3061
tele: 703/697-1481

Teaming (Subcontracting)

One way that small businesses have approached Ada contracts is by teaming with the big guys. A number of Ada contracts have small business set-asides, specifically to spread the Ada involvement to more companies. The question arises, however, how does one convince a larger company to let a smaller, NON-ada smart company, team with them? One answer is to push your strengths. There is a decided difference between Ada Work, and Working in Ada. Very few companies do Ada Work, i.e. their primary language is Ada, and the actual application is secondary. Most have a particular area of expertise, and do that application in Ada. For example, Van Dyke and Associates does a great deal of

Networking business, with some work in Data Bases. We also have done some work in software conversion between platforms. Our thrust has been that we can understand the environment and the problem to be solved, and can build the solution in Ada.

Implement Part of a Project in Ada

If possible, do parts of a larger project in Ada. Yes, this is risky, but if discrete parts of a project are clearly defined, the benefits of doing part in Ada can outweigh the risks. This works only if you have the Ada trained people and environment to handle multiple languages.

Know the Market

Keep up with the market. Most of the periodicals listed below announce pending Ada contracts, as well as those awarded. It is possible to get "a piece of the action" on a large contract. Attendance at conferences and symposia also provide information on "who's doing what" in Ada.

Develop a Product

Develop your own product. There are a number of companies today who started small, and went on to greater endeavors. Often a company has specific knowledge of a problem area, and can build useful software. This can lead not only to sales, but to other areas of working in Ada.

Sources of Information

The following are some resources available to anyone for little or no cost. The resource that cannot be reduced is time, although there are training aids that can help improve the training effort. It will be necessary to spend sufficient time to learn why Ada is different and how to use it. The resource list is by no means all inclusive. The inclusion of a vendor name or product does not imply endorsement.

AdaIC - Ada INFORMATION Clearing House

This one source is perhaps the most inclusive of information resources. The AdaIC is a resource of resources. It is operated by the IIT Research Institute for the Ada Joint Program Office (AJPO). The AdaIC information can be obtained in hardcopy or electronically from the AdaIC BBS or the AJPO host on the Defense Data Network(DDN). The information is available to ANYONE. They publish information on the Ada community's events, working groups, research, publications, and concerns, including a quarterly newsletter. There is no charge for the subscription. Subscribe or request information by writing to:

Ada Information Clearinghouse
c/o IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706-4320
or
tele: 800/AdaIC-11 or
703/685-1477.

Books

AdaIC has a significant book list. A few popular books found in many bookstores and often used as a text in academia include:

Programming in Ada
J.G.P. Barnes
Addison Wesley

Software Engineering with Ada
Grady Booch
Benjamin/Cummings

Ada for Programmers
Eric W. Olsen, Stephen B. Whitehill
Reston

Managing Ada Projects Using Software
Engineering
Jag Sodhi
T2b Books Inc.

Developing Software to Government Standards
William H. Roetzheim
Prentice Hall

The above mentioned books are only a small subset of available texts. The limited list is only provided as a sample. The authors recommend a search of all available books such as those listed in the AdaIC book list.

Government produced text resources

The Ada Language Reference Manual (Ada LRM), the ANSI/MIL-STD-1815A, is published by the AJPO and can be obtained for very little cost. It is "THE" Ada manual by which all compilers are judged. It can be obtained from

Naval Publications and Forms Center
5801 Tabor Avenue
Philadelphia, PA 19120-5099
215-697-2000

Another DoD produced manual is DoD-STD-2167A. It spells out all the documentation and software engineering elements that must be considered when developing for government contracts. It can be obtained from the above address as well.

The Federal Government sponsors research on a wide range of topics. Published reports of these activities are available to the general public through the National Technical Information Service (NTIS) and to Defense Department activities and Defense contractors through the Defense Technical Information Center (DTIC). New reports available are often published in the AdaIC newsletter. For information on reports or registration contact:

NTIS (for general public)
U.S. Commerce Department
Springfield, VA 22161
or
tele: 703/487-4650

DTIC (for Defense or Defense Contractors)
Cameron Station
Alexandria, VA 22314
or
tele: 703/274-7633
AV: 284-7633

Periodicals

Ada information is spread out among a variety of periodicals. Some charge a subscription fee, but others can be obtained free of charge. Some examples follow.

Defense Electronics (no cost)
Government Computer News (no cost)
Federal Computer Week (no cost)
Embedded Systems (no cost)
The Software Productivity Consortium
Quarterly (no cost)
ACM SIGAda Ada Letters
Computer Language
Journal of Pascal, Ada, & Modula2
Communications of ACM
IEEE Computer
IEEE Software

Training

The myriad of training possibilities range from self imposed textbook study to commercial training companies. Reasonable and effective training can be obtained somewhere in between. While most commercial training companies charge a significant amount, local SIGs and users groups give reasonable priced tutorials. Introductory courses are available at conferences and symposiums. Academic courses are available at many colleges and universities and often your employer will reimburse you for the tuition.

A popular and low cost option is a self paced tutorial called ADA-TUTR. ADA-TUTR is a shareware product developed by John J. Herro, Ph.D., of Software

Innovations Technology. It is a thorough interactive instruction with "homework" assignments. An Ada compiler is helpful but not required. It runs on PC's, minis, or mainframes. There are several different methods to obtain a copy. An original license and documentation from the author is about \$30. You may find it on a local BBS, try it out, and license your copy for \$25 or you can obtain a trial copy for \$10 from the author. Company licenses are also available. You can contact the author at:

Software Innovations Technology
1083 Mandarin Drive NE
Palm Bay, FL 32905-4706
or
tele: 407/951-0233

Electronic Bulletin Boards

AdaNET

Sponsored by NASA, no charge for an account.
For information, contact:

AdaNET c/o MountainNet
Eastgate Plaza, 2nd Floor
P.O. Box 370
Dellslow, WV 26531-0370
or
Tele: 304/296-1458
or
800/444-1458

Ada Technical Support BBS

Navy Computer and Telecommunications
Command
2400 baud, 8 bits, 1 stop bit, no parity
Data: 804/444-7841
Tele: 804/445-4481 or DSN: 565-4481
SysOp: Dave Parker

Ada Information Clearinghouse(AdaIC) BBS

300, 1200, or 2400 baud, 8 data bits, 1 stop bit,
no parity
Data: 703-614-0215 or 301-459-3865

Professional Organizations

Association for Computing Machinery (ACM)
SIGAda. SIG membership is available to ACM members and non-ACM members. You can find a membership application in the monthly ACM Communications magazine or contact ACM by writing to:

ACM
P.O. Box 12114
Church Street Station
New York, NY 10257

IEEE Computer Society. IEEE Computer Magazine is published monthly, a subscription is included as part of membership dues. Non-member subscriptions are also available. For membership information contact IEEE Computer Society at:

IEEE Computer Society
1730 Massachusetts Ave NW
Washington DC 20036-1903

For subscription information, tele: 714/821-8380

Software Engineering Institute (SEI). A professional organization fostering quality software engineering principles. They have specific requirements that must be met to be granted affiliate status. For information on requirements and services, contact:

SEI
Mark E. Coticchia
Manager of Affiliate Relations
Carnegie Mellon University
Pittsburgh, PA 15213-3890
tele: 412/268-5758

Software Technology Support Center (STSC).

The U.S. Air Force Logistic Center (AFLC) has implemented the Software Technology Support Center (STSC) in supporting activities to assist in improving the quality of software. A principle component of this effort is the ToolBox/PC, a database of various tools from various sources. The data base information is disseminated via floppy diskette to requesting organizations. Any interested participant can contribute to the data base or obtain a floppy containing information from it. Also, the STSC publishes various reports on tools and software. The reports contain evaluations of tools in the data base. To contact the STSC, telephone:

Dawn Timberlake
STSC Customer Service
Commercial: 801/777-7703 or DSN 458-7703
Fax commercial: 801/777-8069 or DSN
458-8069

Conferences

The Annual National Conference on Ada Technology. Ten years strong, 1992 is the Tenth annual conference. The conference director/manager is Rosenberg & Risinger. You can contact them at:

ANCOST
Rosenburg & Risinger
11287 W. Washington Blvd.
Culver City, California 90230
or
tele: 213/397-6338.

Tri-Ada. Sponsored by ACM SIGAda, it is the largest of Ada conferences/conventions and like other conferences, it melds Academia, Industry, and Government. The current director/manager is:

Danieli & O'Keefe Associates, Inc.
Conference Management
490 Boston Post Road
Sudbury MA. 01776

Ada Software Engineering Education and Training (ASEET) Team. Sponsored by the ASEET Team, periodic symposia address the issues faced by academia, industry, and government in providing Ada software engineering education and training. The contact for the most recent ASEET symposium is:

Catherine W. McDonald
Annual ASEET Symposium
Institute for Defense Analysis
1801 N. Beauregard Street
Alexandria, VA 22311-1772
or
tele: 703/845-6626

Compilers

To quote the AdaIC Newsletter, as of August 1, 1991 there were 193 validated compilers available with an additional 93 compilers derived from base implementations (validated by registration). The targeted platforms range from the PC to the large mainframes. The AdaIC newsletter contains a current list of validated compilers and the target platforms. Although not necessarily inexpensive, the PC compilers are the least expensive and are quite capable, excellent for training and/or project development. If formal training is chosen as an avenue, some vendors will include a PC based compiler with their training course. Also, some vendors give a student discount when enrolled in a academic course.

Representative vendors of compilers in the \$100 - \$700 range include AETech, Alsys, Meridian, and RR Software. A full list is available through the AdaIC.

Tools

CASE tools are gaining popularity as a means to more rapidly develop and document ideas. Many CASE tools exist ranging from limited capabilities to very complex and versatile software. Most are expensive for the small business unless provided under contract. Representative vendors include Meridian and Evergreen CASE for CASE tools, and Logicon for DoD 2167A

document preparation. The AdaIC has a CASE tools database available, both through the BBS and via mail.

Conclusion

Yes, the Ada market is there.

Yes, it requires an investment.

No, you do not have to be rich to get into it.

Authors' Biographies

Joseph P. Hoolihan has been a computerist for twenty years, and holds a B.A. in Math from State University of New York at Binghamton, and an M.S. in Computer Science from Johns Hopkins University. He has been active in Ada since 1983, and has served on several standards and review committees. He is the Manager for Ada Programs for J.G. Van Dyke and Associates.

Lindon J. Corbett has twenty years of computer related experience in both hardware and software domains. He has been employed with J. G. Van Dyke & Associates for 9 years. He has a Bachelor of Science in Electronic Technology from Brigham Young University and developed an interest in Ada while completing his Masters in Computer Science from Johns Hopkins University. His current position is Senior Ada Systems Engineer.

Authors' Address:

J.G. Van Dyke and Associates
141 National Business Parkway
Suite 210
Annapolis Junction, MD 20701
301-596-7510 or 410-381-7970

Acquisition Model for the Capture and Management of Requirements for Battlefield Software Systems

Harlan Black, US Army Communications - Electronics Command; Research, Development, and Engineering Center; Software Engineering Directorate

Abstract – This paper describes a model for the acquisition of software intensive tactical systems. It emphasizes Requirements Engineering and was designed to meet the needs of new and unprecedented systems that are large and complex. When applied properly, it should reduce the cost, schedule, and quality risks that have been associated with these types of procurements.

1. INTRODUCTION

The following problems have traditionally affected cost, schedule, and quality of large-scale ada-based acquisitions: Solicitation and award of a Full Scale Development (FSD) contract with incomplete and/or ambiguous requirements; delayed requirements definition and documentation; the appearance of contractual relationships that encourage requirements to increase; and dynamic operational environments where requirements continue to change.

The US Army Communications - Electronics Command (CECOM) Research, Development, and Engineering Center's Software Engineering Directorate (SED) is responsible for the post-deployment software support for all communications related systems, including command and control. In early 1991, SED proposed an acquisition model which, when applied properly, should reduce the risks associated with these types of procurements. Subsequent to this research, additional insight provided a motivation to refine this model. This paper describes the revised model.

The term 'acquisition model' denotes a management process. It is a process of obtaining (acquiring) technology. Appropriate technologies would be

utilized within this process, but the model, itself is not a technology.

The intended use of this model is the Army Project Manager (PM), the one responsible for providing a system to the soldier in the field, as specified by the soldier's representative.

The primary focus of this model is placed upon system requirements. "In every software project which fails to meet performance and cost goals, requirements inadequacies play a major and expensive role in project failure¹." Project impact from requirements related problems increases drastically with the time of detection. Figure One illustrates the elusive nature of apriori requirements specification. This model stresses the importance of carefully engineered system requirements. It has long been observed that the delineation of requirements is often incomplete, inconsistent, and frequently specified at varying degrees of detail, all of which significantly contribute to the risk of the development. FSD contracts have been awarded with incomplete, erroneous, inconsistent, and ambiguous requirements, as the time and effort needed to improve upon requirements definition are frequently underestimated. These errors are frequently not discovered until much later in the development and acquisition process, resulting in cost and schedule growth. In addition, there have been systems for which the specification of user interface and interaction detail was delayed until the critical design review, making changes and improvements very costly in dollars and schedule.

The model is proposed within the context of DOD-STD-2167A and can be tailored to apply to a wide range of acquisitions. The intent of this model is to characterize process model strategies for

Requirements Engineering, not to fully specify every detail for its implementation.

Finally, the model is organized and defined in terms of six risk reduction strategies. These strategies have been recommended by numerous studies and workshops (reference ² through ⁵).

While this model should reduce the quantity and severity of requirements related problems, it is not envisioned that they will or can be eliminated. We will always have valid needs to change requirements, from such reasons as advances in technology, changes in enemy tactics and capabilities, changes to external systems which must be interfaced with, and insight gained during the system implementation.

2. REQUIREMENTS ENGINEERING

It is not sufficient to write requirements. Requirements must be **engineered** and **managed**. This model strongly suggests the early designation of a team or effort that is responsible for engineering the system's requirements, the Requirements Engineer (RE). Requirements Engineering is the process of applying engineering disciplines to requirements definition and management.

For the purpose of this paper, there are two modes of requirements development and evolution: pre and post solicitation. Pre-solicitation requirements define what will be acquired. Post-solicitation requirements reflect changing project needs and maintain fielded-product relevance.

Also, for the purpose of this paper, there are two types of initial system fieldings: quick fielded and delayed fielded. Assuming rapid and robust prototype development capabilities, quick fielded systems have minimum delays for hardware development. An example of a quick fielded system is a Command and Control system for maneuver control, whose incremental versions can readily run on commercially available hardware. Delayed fielded systems require extensive hardware development prior to the initial fielded version. An example of a delayed fielded system is the operator interface for a yet to be developed tank. While prototypes can be simulated, it takes years for the hardware to be developed, the software integrated, and system fielded use for end user feedback.

This model recognizes the need for Requirements Engineering for every phase and every type of system. It is based upon a realization that it is unrealistic to expect full requirements definition prior to the solicitation of the typical non-precedented large-scale software system. Organizations, technologies, interface requirements, and threats are in a constant state of change, greatly impacting the ability for the requirements to be clearly state up front, prior to the acquisition. Frequently, the requirements are so complex and interrelated that it is not humanly possible to specify them prior to the solicitation.

For the pre-solicitation requirements, the PM must either specify a system with features to be determined or the PM must devote significant resources to capture requirements detail and reduce acquisition risk.

The former approach has worked well for the US Army's European tactical command and control system⁶. This effort "combines rapid prototyping based upon minimum, or thin, specifications with frequent operational deliveries to the field ⁶." The project's philosophy is to "build a little, test a little, use a little ⁶." However, this approach appears to be only feasible for quick fielded systems. For delayed fielded systems, this model suggests that the RE be given significant resources to engineer the pre-solicitation requirements to get them to at least the level of the System Segment Specification (SSS).

For post-solicitation requirements refinement, either the RE or the FSD can work with the user or user representative. There is an advantage in having the FSD contractor fulfill this role. It gives contractor personnel the opportunity to come up to speed and keep on top of the evolving requirements with first hand experience with the user's problem. A disadvantage is the risk of the appearance of a conflict of interest, as requirements development can have a positive impact on future business for the contractor. Either way, the RE must, at a minimum, keep track and manage the requirements and their impact to the project.

This model recommends that the RE be involved with requirements related issues throughout the lifetime of the project, not just during its early stages. During system development, the RE should interact with the user or user's representative regarding proposed changes to the baseline. The RE should interact with

end users after initial system fielding to independently gain their feedback. Relevant activities include: risk and feasibility analysis; trade-off studies; requirements change impact analysis; tracing requirements between documents; maintaining the consistency of requirements documents; verification that requirements are being met by the developer; and supporting the PM during reviews and audits.

3. THE ACQUISITION MODEL

This acquisition model stresses Requirements Engineering, emphasizing techniques for requirements definition and change management.

The model is defined in terms of the following six strategies for risk reduction:

- Designate a Requirements Engineering effort which applies Requirements Engineering techniques from the early project phases and on.
- Plan to develop systems in an incremental, evolutionary manner.
- If the Requirements Engineer is a contractor, contractually decouple the RE effort from the FSD effort.
- When applicable, establish a Functional Baseline (FBL) with an approved SSS prior to the solicitation and make the SSS a part of the solicitation package.
- Document the user interface and interaction in the SSS, together with system testing information.
- Provide structure for the relationship and interaction between the user and the FSD contractor for all requirements related matters.

The following subsections elaborate upon these strategies.

3.1 Designate a Requirements Engineering effort which applies Requirements Engineering techniques from the early project phases and on.

One can view the RE as having a role that is similar to an architect of a building project⁷. The RE must be under the control and direction of the PM. As this function is highly technical and system oriented, it

may be appropriate for the project's system engineer to be assigned the lead responsibility. The work, must begin no later than the initial drafts of the user requirements documents, early on in the project and before commitments by Government and contractors are made.

The RE must wear many hats. To the PM, the RE is a consultant on requirements and their impact. If the RE is responsible for specifying post-solicitation requirements, the RE is the system developer to the user, exploring the feasibility and impact of user requirements and suggesting options, when there are trade-offs.

Staffing the RE team is a non-trivial and critical task. It is most advantageous for the Government to have its own personnel perform this function directly, not through a contractor. The Government, itself, must be the one who is the most aware of what it needs, the system requirements. However, this may not always be feasible due to personnel constraints. It may therefore require contractual support.

The PM must carefully assess the requirements for the Requirements Engineering effort and then monitor it carefully. Just as with the FSD effort, the risk of requirements proliferation exists during Requirements Engineering. Unlike the FSD effort though, this effort is on a much smaller scale, reducing risk and impact.

3.2 Plan to develop systems in an incremental, evolutionary manner.

Our battlefield systems are often too dynamic and/or complex to field successfully in a single release. It is, therefore, very difficult to plan for a system's development in one release or block.

Plans for the development should call for incremental releases of the system. It is recommended that users prioritize their requirements, listing and rating them by need and by certainty. Requirements that are certain, well understood and that are critical to user/system functionality should be met in the initial release. This initial system must be useful to the user, providing essential capabilities, albeit it is not everything that is needed.

Requirements for subsequent releases can become separate options on the FSD contract or they can be

separate procurements, depending on the system's acquisition approach.

References for examples of successful evolutionary system developments are provided ^{6,8,9}.

3.3 If the Requirements Engineer is a contractor, contractually decouple the RE from the FSD effort.

Requirements are a major determinant in acquisition cost and schedule. They should therefore be engineered by an independent agent. A RE contractor should be precluded from the FSD competition and subcontracting.

3.4 When applicable, establish a FBL with an approved SSS prior to the solicitation and make the SSS a part of the solicitation package.

For delayed release systems, this model recommends that the RE write the SSS and conduct the System Requirements Review prior to the solicitation. The approved and validated SSS would then come under Government configuration control and become part of the FBL. The SSS should become a part of the solicitation package. By doing so, we will know what we are buying and bidders will know what we really want.

This approach does not eliminate the possibility of changing the requirements during the solicitation period and during the development with controlled revisions of the SSS. However, it does reduce some of the opportunities for changes with serious impact to occur.

Requirements for subsequent releases must also be documented in the SSS. They can be stated in separate appendices. At this point they do not need the great detail of the initial release's requirements.

3.5 Document the user interface and interaction in the SSS, together with system testing information.

Cost and schedule can be significantly compromised when user interface and interaction details are not agreed upon in a timely manner. Section 3.2.3 of the SSS format describes the interfaces with external systems. This is an ideal place to provide detail on the man-machine interface and interaction from the user-perspective of the system. For delayed release

systems, this model recommends that this be done by the RE prior to the solicitation.

It should be noted that section 4.0 of the SSS deals with provisions for quality assurance. Test case requirement coverage and general system test philosophy should be specified by the RE in this section. Additionally, the RE may be asked to specify the system requirements test plan and cases in separate documents. For some developments, it may be appropriate for the RE to support or actually perform the testing.

3.6 Provide structure for the relationship and interaction between the user and the full-scale development contractor for all requirements related matters.

As mentioned previously, the relationship between the FSD contractor and the user can unknowingly contribute to requirements growth. This model recommends that the user/ FSD interaction be restricted to the point where there is no risk of the appearance of a conflict of interest. For example, when the RE is responsible for specifying post-solicitation requirements, the contractor should be restricted from picking up the phone and suggesting new requirements directly with the user. Rather, the user and the contractor should interact with the PM and RE.

In any case, the user representative should be an active participant in the system's formal reviews. These reviews provide a formal and controlled environment for user-developer interaction. Understandably, interactions such as end user evaluation at the contractor site should not be precluded.

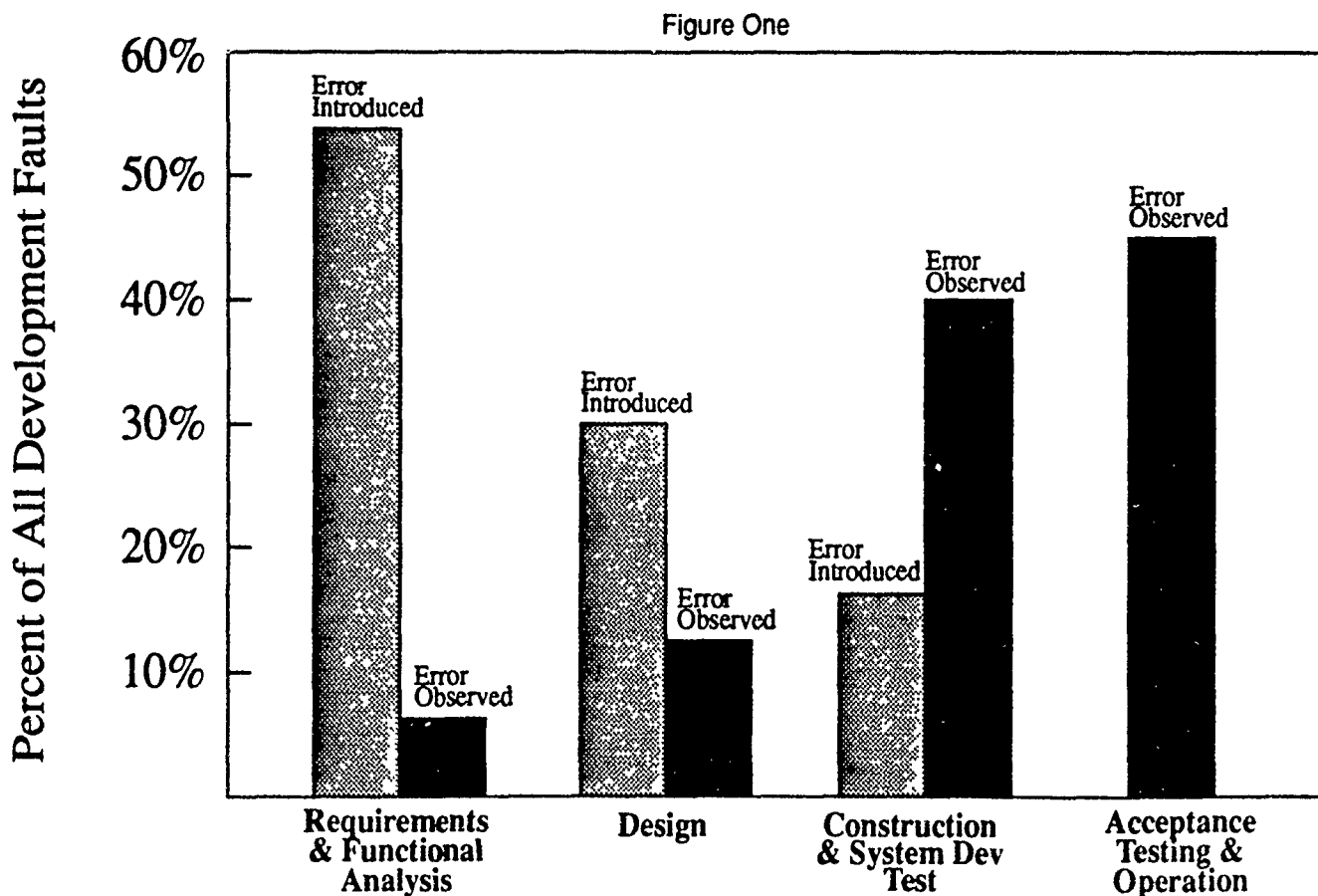
4. CONCLUSION

The acquisition model presented in this paper proposes a small change to the current acquisition process. Current policies, regulations, and standards do not preclude the implementation of this model, but they do not encourage it, either.

A step-by-step approach is planned to gain recognition and acceptance of the strategies in this model. Our near term goal is for the model to be implemented and validated on a CECOM pilot project.

REFERENCES

1. Afford, M. W. and Lawson, J. T., "Software Requirements Engineering Methodology (Development)", USAF Technical Report RADC-TR-79-168.
2. Beam, W. R. et al, "Adapting Software Development Policies to Modern Technology," Washington, DC: National Academy Press, 1989.
3. Black, H. et al, "The Technical Cooperation Panel (TTCP) Requirements Engineering and Rapid Prototyping Workshop Proceedings," US Army CECOM Center for Software Engineering Technical Report C-0103400000100, May 1990.
4. Beam, W. R. et al, "Adapting Software Development Policies to Modern Technology," Air Force Studies Board, 1989.
5. Hess, J. A. et al, "Report of the AMC Software Task Force," US Army Materiel Command, February 1989.
6. Giordano, F., Wong, B., and McCollum, L, "Rapid Development Speeds Path for Command System" Signal Magazine, April 1991.
7. Smirall, G. E., "Requirements Engineering and Ada," Proceedings of the 6th National Conference on Ada Technology, March 1988.
8. Bersoff, E., Davis A., "Impacts of Life Cycle Models on Software Configuration Management," Communications of the ACM 34, 8 (August 1991), 104 - 117.
9. Davis, A., "Operational Prototyping: The POST Story," Submitted to IEEE Software, 1991.



Source: "Software Engineering," Ramamoorthy, et al., IEEE Computer, 10/84

Alternative Documentation and Review Practices

Raymond J. Menell and Lisa A. Heidelberg

U.S. Army CECOM Software Engineering Directorate

Fort Monmouth, New Jersey 07703

Abstract: This paper provides ideas and guidance for implementing practices for tailoring the DOD-STD-2167A (2167A) Software Development Process in order to: 1) reduce excessive documentation and reviews thereby reducing cost; 2) increase visibility and control of the product; and 3) shorten life cycle development time. The information in this paper has been developed by the Fort Monmouth U.S. Army Communications-Electronics Command (CECOM) Software Engineering Directorate (SED) as an outgrowth of CECOM/Industry Documentation Task Force (DTF) findings. In general, these findings have pointed to an overreliance on the part of CECOM on plain language documents and formal reviews. The principal recommendation is to place more emphasis on the technical staff reviewing projects and less emphasis on the documentation.

1. Introduction

Current documentation and review practices are inefficient and costly as they require the developer to continually translate their internal representations of software to documents and formal viewgraph presentations that are highly structured, formatted, and text-based.

Furthermore, the software documents required by the standards are not the best medium for adapting to change or performing traceability. The reason is that the customer's requirements are being repeatedly modified and/or new requirements are being added that impact the design, code, and test documentation. Also, design is typically not a straightforward top-down approach. Design trade-offs and decisions by a developer are continuously being made during development causing changes to software designs and documents. Furthermore, modern design methods used by a developer require that multiple iterations of design be performed before either the preliminary or detailed designs become stabilized.

The alternative practices being implemented at CECOM involve the use of a Software Review Team (SRT). The team uses software development information rather than documentation to evaluate the completeness and adequacy of the system. The team is composed of software engineers from CECOM and/or support contractors who have the ability to read

and understand the developer's software process, method, design representation, and Ada implementation. An SRT does not need to rely on text-based, plain language documents, and formal reviews. Therefore, it is possible to eliminate the Software Design Documents (SDDs) and the Interface Design Documents (IDDs) during the development stages. The team uses the developer's Software Development Plan (SDP) to understand the developer's plans and process, method, and procedures for developing the system. Additionally, the SRT uses the developer's Computer Aided Software Engineering tools to facilitate the review of requirements and design representations. Also, the SRT accesses the developer's configuration management facility to trace requirements through design, code, and test. The SRT performs its reviews at the contractor's facility during technical interchange meetings. However, the ideal situation for the CECOM SRT would be to perform routine reviews at their own worksite.

The benefits of implementing these practices include: 1) the cost for documentation and reviews would be reduced; 2) CECOM CSE would be able to provide timely and accurate information to PEOs or PMs; and 3) CECOM CSE would have a resource of people who can provide transitional support and PDSS.

2. Background

The insight into the problems and solutions arose from two sources. The first was from the CECOM/Industry Documentation Task Force (DTF), which was initiated by CECOM at

SoftCon 89. The result was a report that defined problems and recommended improved practices. The second source was primarily on-going interactions with CECOM PEOs and PMs. These interactions include implementation of the DTF approach in actual Procurement Data Packages and "experiments" to help understand how to implement these practices. The experiments are partial implementation of the practices.

3. Problems Addressed

The CECOM-sponsored SoftCon 89 Conference, held on 24, 25, and 26 January 1989, produced numerous papers, which collectively made the following assertions:

- a. An excessive volume of documentation is called for by the Government.
- b. Excessive time, effort, and money are spent for software documentation preparation, review, and maintenance.
- c. Excessive cost of documentation is incurred by the contractor and the Government.
- d. The utility of the produced documentation is questioned by system users and maintainers.

The Documentation Task Force in response to the SoftCon 89 Conference showed that the numerous changes in requirements and design were at the root cause of these problems. Since requirements are inevitably being modified or new ones are being added, changes are needed to associated design and test documents. Also, design trade-offs and decisions are continually being made. In particular, during the developer's design activity, the software design may require

numerous iterations of redesign before the design becomes stabilized. Therefore, design change causes documentation to be expensive because the developers or contractors need to transform their abstraction of a design to a "plain language documentation"; in other words, documentation that is formatted and text-based. Plain language documents do not adapt to change very well. The change also impacts the traceability of requirements to design.

Some contractors have special organizations that write and review documentation and spend numerous hours presenting the information to CECOM at formal reviews. This is very costly and inefficient. Often a plethora of documentation arrives a week or two before the formal review due to the documentation being changed. This gives CECOM reviewers very little time to read documentation prior to formal reviews. Often, by the time CECOM has reviewed the documents, the design changes. One of the by-products is that the documentation production cycle may create adversarial relationships between the contractor and CECOM.

Another problem with the Government review approach is that some reviewers have not been trained in modern software development practices. Oftentimes reviewers do not understand the developers design representation and require that the design representation be transformed into a plain language design documentation. Consequently, developers usually question the reviewer's comments on the design. Some developers have commented: "Why are nonsoftware people commenting on

software design if they are not trained in software design?" and "What value are nonsoftware individuals adding to the software development effort?" However, there is a high level of design that may be needed by nonsoftware type people. For example, logistics and training professionals may need an abstraction of the overall design. Therefore, these professionals should be presented with this level of information. This will be discussed in Section 4.3 of this paper.

The tailoring of DOD-STD-2167A documentation appears to be another problem. Software Engineers at CECOM currently are using the LOGICON's Tailor/2167A tool and/or MIL-HDBK-287, 2167A Tailoring Guide. The tool and Tailoring Guide both are useful. The LOGICON tool automatically manages what the software engineer tailors from the Data Item Descriptions (DIDs). The Tailoring Guide provides insight into other process models (besides the waterfall model) and its appendixes have examples of how the software engineer can apply the DIDs for three modes of development: 1) Concept Demonstration; 2) Full Scale Development; and 3) Production Systems. However, neither the tool nor the Tailoring Guide provides the Software Engineer with the training or insight necessary to make the software engineering decisions that are useful to reduce excess in the Contract Data Requirements List. Software Engineers at CECOM complained about how little of the documentation was reduced when they applied the Tailoring Guide's recommendations to Full Scale Development. For example, when guidance from the Tailoring Guide was used in

the tailoring of SDDs, the result was the SDDs in almost their entirety. Current tailoring guidance does not solve the real problems of tailoring the SDDs because: 1) the level of information requested by Government for the SDDs is too detailed and/or premature and; 2) design information is volatile and difficult to keep updated during development. In other words, there is very little understanding of the timing and behavior of software development information and how to implement it within the CDRLs to minimize documentation. Also, the 2167A paper documentation is not conducive to understanding the design.

The following guidelines are an attempt to overcome these problems.

4. Guidelines to Implement the Alternative Documentation and Review Practices

Specific guidance sections are provided regarding how to implement the following alternative practices:

- Establish a Software Review Team (SRT)
- Tailor the 2167A Documentation
- Use a High-Level Software Design Document (HLSDD)
- Reformulate PDRs and CDRs
- Adopt a Content-Specific SDF
- Incorporate Electronic Methods of Review.

Following each guidance section is a discussion section to promote understanding and provide

rationale to the user.

4.1 Establish a Software Review Team (SRT)

Guidance:

- Establish an SRT in order to be less dependent on documents and formal reviews. To do so, develop a charter or sponsoring agreement with the CECOM Project Manager and SRT that defines the responsibilities and the resources required for the SRT to perform its duties. The SRT may be chartered directly by the PM or PEO. The SRT may also be formed by a sponsoring agreement within a chartered group, such as the Computer Resources Working Group (CRWG).
- Define the relationship between the SRT and developer in the SOW.
- Have the developer provide training to the SRT on the use of their development software engineering environment and processes.

Discussion:

An SRT is composed of Software Engineers from CECOM and their support contractors who have the ability to read and understand a developer's software process, method, design representation, and Ada language implementation. Some members of the SRT also understand the specific application domain. Members of an SRT are able to use the developer's Software Development Plan (SDP) to understand the process, methods, practices, and tools. The SDP also provides the SRT with the developer's schedules; therefore, the SRT is able to determine when the developer plans to complete life-cycle objects during the

development in order to review them. The SRT, being technically competent, can use the developer's CASE tools to access, understand, and trace requirements, design representations, Ada code, and tests. Also, the SRT is able to utilize information located in the developer's SDFs. Metrics may be used by the SRT to assist in focusing in on problem areas. Specific training in the developer's software engineering environment, processes, and/or in the application domain may be required for the SRT. Many developers already have training courses for their new employees and SRT members can take the same courses.

The team members perform analysis at the contractor's facility during technical interchange meetings. However, an alternative approach is for the CECOM SRT to perform reviews on-line through electronic communication methods.

The SRT can be dynamic and may need to utilize additional resources or special skills as required. For example, if Ada run time kernel issues need to be addressed, an appropriately knowledgeable individual should be employed. However, it is recommended that a group of core people remain on the project during development.

Members of the SRT should be part of the proposal evaluation team that would determine if a proposal satisfies the needs of the SRT. As part of the proposal evaluation it is very important that the potential developer define their approach and facilities to develop and test the software system.

The transition of information from the developer, through the software review team, and to the PM

is very important. A CECOM SRT evaluates and analyzes information from Technical Interchange Meetings (TIMs), SDFs, and CASE-tool generated representations of the design. Once the software development information is evaluated and analyzed by the SRT, it is transferred as information to the Project Manager through briefings, reports, and memos.

A benefit of using an SRT during development is that CECOM personnel are becoming knowledgeable of the software and are better able to provide support for transition and PDSS.

One concern that has surfaced regarding the use of an SRT is whether or not the developer will allow an SRT to come on-site. The relationship between the developer and the SRT and the responsibilities of the developer must be defined in the Statement of Work (SOW). One of the major findings of the Documentation Task Force was that the developers in the group said they would open their software development files and environments to individuals if they were trained in software engineering and Ada. They would prefer this to generating plain language documentation and interacting with people who do not understand the software development representations. In order to insure the SRT is successful, the developer must be contractually bound to support the SRT.

4.2 Tailor the 2167A Documentation

Guidance:

- In the SOW, eliminate the delivery of the SDDs and IDD during development, and stipulate that the SDFs be delivered

incrementally with specified contents. For additional information on SDFs, please see Section 4.5 of this paper.

- Specify in the SOW that either the "as built" Software Product Specification (SPS) documents or the contents of the SDFs be deliverable. If the SPSs are specified as deliverables, then the detailed design portion of the SDDs can be deleted in many cases.
- Specify that requirements traceability tables be provided by the contractor in electronic media.

Discussion:

The SRT reviews the software development information utilizing the contractors CASE tools and SDFs instead of 2167A documents. Once the design is stabilized, the design representation can be put into a plain language "as built" documents, if desired. The "as built" documents can be used for PDSS in the form of Software Product Specifications (SPSs) along with the IDD. Also, it may not be necessary to include the detailed design section of the SDD in the SPS since the Ada code would provide the level of detail to understand and maintain the design. However, if the developer's SDFs are specified as deliverable, there may be no need for SPSs.

Electronic generation of traceability tables is recommended. Traceability is required between requirements and design, between the requirements and test procedures, and between design and source code. When working documentation changes, the traceability between tables must change to reflect the current system.

Also, CASE tools working with SDFs stored in an on-line database is a highly recommended practice for usage on a project that implements this practice. The tools are needed to construct, maintain, and access the repository of working documentation information; to provide the traceability relationships needed among the life-cycle objects (requirements, design, source code, test cases, etc.); and to generate the formal documentation needed at the conclusion of the software development. Automated support is also needed to present working documentation in a usable form for use by the developers and the SRT.

4.3 Use a High-Level Software Design Document (HLSDD)

Guidance:

- Have the preliminary design activity be at the total software system level via the HLSDD rather than at multiple CSCI levels. Please see Section 4.4 of this paper on reviewing the HLSDD.

Discussion:

The level of detail called for in the SDD and IDD at Preliminary Design Review (PDR) is called for too early in the systems life cycle. Information such as memory and processing time allocation, control and data flow, local Computer Software Component (CSC) data, timing and sequencing, and error handling are often not available. Therefore, why force a contractor to produce (or fake) this information in formal documentation and reviews if it is going to change or needs to change in order to generate a quality system? Part of the problem is that design is not a complete

top-level process but also requires a bottom-up thought process. In modern design methods, it is difficult to identify objects at the early stages of the design process. Sometimes a developer requires numerous iterations to establish a satisfactory design object. Under 2167A, a developer would be required to redo the design, planning, and traceability documentation for each design iteration.

There is need for a level of information or design that may be useful to individuals who are not trained in software engineering and Ada. However, even for the technical person a high-level type of document is necessary because the total software design sometimes gets lost within the Computer Software Configuration Item (CSCI) parts. Therefore, a high-level design review and document may be needed to help see the whole software design and to give those without a software background insight into the design.

4.4 Reformulate Preliminary Design Reviews and Critical Design Reviews

Guidance:

- Hold a single PDR based on the HLSDD that addresses the entire software system down through CSCI identification and functionality.
- Specify that design reviews become a progressive series of reviews carried out by trained and qualified software review team on the developer's working documentation (e.g SDFs). The reviews are performed during Technical Interchange Meetings or more ideally at CECOM through the use of CASE tools and/

or electronic communications. Another implementation of this guidance is to split the CDRs into TIMs and a Management Sessions.

Discussion:

The traditional PDRs and CDRs, according to MIL-STD-1521, are replaced in part by the evaluation of "Working Documentation" via TIMs between the developer and the CECOM SRT. Working Documentation resides in the developer's Software Engineering Environment (SEE) and SDFs and is a dynamic electronic representation of the status of the software development.

4.5 Adopt a Content-Specific SDF

Guidance:

- Specify the content of the SDFs to supplant 2167A documentation and accept the SDFs "as built" in incremental deliveries.

Discussion:

The SDFs serve as living documents during a 2167A development. It is the contractor's Working Documentation that is called for by the DTF report. This means that the requirements design and test information it offers is the most current available from any documentation sources within the project. While this guidebook calls for tailoring 2167A practices, the usage of SDFs (which is a 2167A practice) is an excellent form of documenting and reviewing the software development.

Further, by specifying the contents of the SDF in the areas of requirements traceability, interface

design, and test, the SDFs will then prove to be a superior replacement for such 2167A documentation as SDDs, IDDs, and SPSs.

4.6 Incorporate Electronic Methods of Review

Guidance:

- Incorporate into the SOW provisions allowing for electronic methods of access from a remote site and on-site at the contractor's facility on a read only basis to a specific directory area of the contractor's software engineering environment. These provisions should allow for review of the following:

- Ada Design Language
- Requirements traceability
- CASE design representations
- Ada source code
- Ada test cases/results

Discussion:

Travel to and from contractor's site for review of developmental status is costly both in time and money. Meanwhile, when SRT members representing the PM are on site there are problems associated with reviewing current stages in the product under development. Electronic methods of review allow for cost effective access to critical information associated with the products development.

For example, most contractors use requirements traceability tools, Ada Design Language, Ada Language, and Ada test procedures, all of which are available in the host computer development environment. Electronic methods of review

provides more timely information on the status and risk areas of projects.

5. Afterword

The guidance given in this paper arises from the collective minds of Fort Monmouth CECOM and Industry concerning Ada software acquisitions. Nevertheless, partial implementations of the guidance have occurred only recently. Although the practical results of this guidance are proving to be very effective, it is hoped that with further experience in implementing these approaches and studying their consequences, the approach of this Guidebook might be improved and refined.

Raymond Menell is a Software Engineer with the U.S. Army, CECOM Software Engineering Directorate, Software Technology Division, Fort Monmouth N.J. He is currently involved in improvements in the use of software methods and tools by CECOM for software review and Post Deployment Software Support. He received a BS and MSCS from Monmouth College, N.J. and is currently completing his studies for an MS in Software Engineering at Monmouth College.

Lisa Heidelberg is a Software Engineer with the U.S. Army, CECOM Software Engineering Directorate, Tactical Communications Branch, Fort Monmouth N.J. She is currently involved in the acquisition, and system and software design of a Mission Critical Defense System. She received her BS in Computer Science from Trenton State College, N.J. and her MS in Software Engineering from Monmouth College, N.J.

USING PETRI NET REDUCTION TECHNIQUES TO DETECT ADA STATIC DEADLOCKS

P. Rondogiannis

M. H. M. Cheng

Department of Computer Science
University of Victoria, Victoria, B.C., Canada V8W 3P6

Abstract: Recent research in static analysis of Ada programs is based on the proper manipulation of the Petri net representation of the programs using techniques from Petri net theory. We present an algorithm for detecting static deadlocks in Ada programs, which is based on Petri net reduction techniques. The proposed approach has been influenced by current research in Process Algebras, and has, for this reason, a clearer theoretical basis than other existing techniques.

Index Terms: Ada, Deadlock, Petri Nets, Reduction Techniques, Ada Nets, Static Program Analysis.

1. Introduction

Over the past few years, a number of techniques have been proposed for the detection of deadlocks in Ada programs. These techniques are based on static analysis, that is, analysis that is performed on a model of the program without requiring test executions. Taylor¹, proposed a general-purpose algorithm for the analysis of Ada programs that is based on the generation of the so called *concurrency histories* of the program. As this method is in fact based on state enumeration, it is inherently inefficient. More recent work^{2,3}, is based on the representation of the program by a Petri net (or Ada net). The objective of such a representation is to apply techniques from Petri net theory in order to reduce the complexity of the state enumeration approach.

In this paper, we suggest that the analysis of

the Ada nets can be significantly eased if the nets are simplified using *reduction techniques* from Petri net theory. The proposed approach is based on the observation that Ada nets carry redundant information which is useless during deadlock detection.

The rest of this paper is organized as follows: Section 2 gives a basic introduction to Petri nets. Section 3 describes the reduction techniques we are going to use. Section 4 contains a number of theoretical results that extend the proposed reductions so that they can be applied for the detection of deadlocks in Ada programs. Section 5 presents the deadlock detection algorithm, and section 6 illustrates the algorithm by an example. Finally, section 7 presents the main results of the paper as well as related work.

2. Definitions

Petri nets are a tool for the study of systems. Petri net theory allows a system to be modelled by a Petri net, a mathematical representation of the system. Analysis of the Petri net can then, hopefully, reveal important information about the structure and dynamic behaviour of the modelled system. In the literature, several different versions of Petri nets have been proposed, depending on the level of detail at which one wishes to describe a specific system. The notation adopted here has turned out to be very useful in relating Petri nets with concurrent programming languages.

Definition 1 A *Petri net* (or simply *net*) is a structure $R = (Pl, T, M_0)$ where:

1. Pl is a possibly infinite set of *places*.
2. $T \subseteq \Delta(Pl) \times \Delta(Pl)$ is a set of *transitions*.
3. $M_0 \in \Delta(Pl)$ is the *initial marking*.

Here $\Delta(Pl)$ denotes the set of all non-empty, finite subsets of Pl . For a transition $t = (I, O)$, its *preset* or *input* is given by $pre(t) = I$ and its *postset* or *output* is $post(t) = O$. Similarly, for $p \in Pl$, $pre(p)$ denotes the set of transitions that have p in their postset and $post(p)$ is the set of transitions that have p in their preset. Petri nets are usually represented graphically in the following way: places $p \in Pl$ are represented as circles \bigcirc with their name p outside, and transitions $t = (\{p_1, \dots, p_n\}, \{p_{n+1}, \dots, p_{n+m}\})$ are represented as bars $|$ carrying the label t outside and connected via directed arcs to the places in $pre(t)$ and $post(t)$. The initial marking M_0 is represented by putting a dot \bullet (or *token*) into the circle of each place in M_0 . The dynamic behaviour of a Petri net, is accomplished through the execution of transitions. Although in the initial marking of a Petri net only single tokens are allowed for each $p \in M_0$ (i.e. M_0 is a set), the execution of transitions may result in places having more than one tokens. To describe this situation, the notion of a *multiset* is used, i.e., a set where multiple occurrences of elements are allowed. Then...

Definition 2 A *marking* M of a Petri net $R = (Pl, T, M_0)$ is a multiset over Pl .

Let \sqsubseteq , \sqcup and $-$ denote multiset inclusion, union and difference respectively. Then the execution of a transition is defined as follows:

Definition 3 Let $R = (Pl, T, M_0)$ be a net, $t = (I, O)$ a transition of R and M be a marking of R . Then:

1. Transition t is *enabled* at M if $I \sqsubseteq M$.
2. If enabled at M , the *execution* of t transforms M into a new marking M_1 of R , and $M_1 = (M - I) \sqcup O$. In symbols: $M \xrightarrow{t} M_1$.

A marking that can be reached by successive executions of transitions is called a *reachable marking*. Formally:

Definition 4 A *reachable marking* is a marking M for which there exist intermediate markings M_1, \dots, M_n and transitions t_1, \dots, t_n with $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} M_n = M$.

Definition 5 A net R is *safe* if and only if in every reachable marking, the number of tokens per place is either zero or one.

The *reachability graph* is a tool that has been used for the analysis of Petri nets. Intuitively, a node of the reachability graph corresponds to a reachable marking of the Petri net, and an edge between two nodes corresponds to a transition execution which transforms one marking into another. Formally:

Definition 6 The *reachability graph* of a Petri net $R = (Pl, T, M_0)$, is a graph $RG = (V, E)$ where $V = \{M : M \text{ is reachable from } M_0\}$ and $E = \{(M_1, M_2) : M_1, M_2 \in V \wedge \exists t \in T, M_1 \xrightarrow{t} M_2\}$.

The reachability graph of a safe Petri net can be used to detect deadlock markings of the net, i.e., states where no further transition can execute. The following two theorems suggest how:

Theorem 1 The reachability graph of a safe Petri net is finite.

Theorem 2 A node of the reachability graph of a Petri net that has no outgoing edges (*sink node*), indicates a deadlock marking of the Petri net

We can now outline the proposed deadlock detection technique: Given an Ada program P , we initially construct the corresponding Ada net R . The straightforward approach would be to consider the reachability graph of R , and search for sink nodes that would indicate deadlock states of the program. However, such an approach is equivalent to exhaustive state enumeration. Instead, we use reduction techniques on R in order to get a simpler net R' which has a smaller reachability

graph. The crucial property of the reductions we use is that they preserve the deadlock information of the initial net. The proposed reductions are presented in the next section.

3. Reduction Techniques

In order to ease the analysis of Petri nets, a number of transformations have been proposed in the literature^{4,5,6} which simplify the net while preserving some of its important properties. In general, different transformations preserve different properties of the initial Petri net, and aim at different goals. The transformations we have adopted preserve safeness and deadlock freedom and allow, as we are going to show, an elegant treatment of deadlock detection in Ada programs. More specifically, we use *post-fusion* and *pre-fusion* of transitions, as well as *elimination of redundant places*.

Before giving formal definitions, we discuss the intuition behind the transformations. Elimination of redundant places consists of the removal of places whose marking is always sufficient to allow executions of transitions connected to them. This kind of transformation does not cause any modification to the behaviour of the net. On the other hand, fusions of transitions have been defined in order to make indivisible some transition sequences representing actions which may occur more or less at the same time. They are based on the fact that *it is not mandatory for a transition to execute as soon as it can execute*. Although the formal definitions of the transformations that are given below seem complicated, their intuition can be easily understood by the accompanying figures.

Definition 7 Let $R = (Pl, T, M_0)$ be a Petri net. A non-empty subset F of T is *post-fusable* with $h \in T$ if and only if there exists a place $p \in Pl$ such that the following conditions are satisfied:

1. The only input of every $f \in F$ is p .
($\forall f \in F, pre(f) = \{p\}$)
2. Place p is not an output of any $f \in F$.
($\forall f \in F, p \notin post(f)$)

3. There exists a transition in F that has at least one output place.
($\exists f \in F, |post(f)| > 0$)
4. Place p is not an input of h .
($p \notin pre(h)$)
5. Place p is an output of h .
($p \in post(h)$)
6. Except for h and the transitions belonging to F , no other transition is connected to p .
($\forall t \in (T - (\{h\} \cup F)), p \notin pre(t) \wedge p \notin post(t)$)
7. Place p holds no token initially.
($p \notin M_0$)

Whenever the above conditions hold, the Petri net can be modified according to the following definition:

Definition 8 Let $R = (Pl, T, M_0)$ be a Petri net, and let $F \subseteq T$, $h \in T$ and $p \in Pl$ satisfy the conditions of Definition 7. Then, the system resulting by the post-fusion of F and h is $R' = (Pl', T', M_0)$, with.

1. $Pl' = Pl - \{p\}$
2. $T' = (T - \{h\} - F) \cup F'$ with F' defined as:

$$\begin{aligned} &\{hf_i : f_i \in F \text{ and} \\ &pre(hf_i) = pre(h), \\ &post(hf_i) = (post(h) - \{p\}) \cup post(f_i)\} \end{aligned}$$

where hf_i denotes the concatenation of h with f_i .

Definition 8 is illustrated in Figure 1(a) for the case $F = \{f\}$ and in Figure 1(b) for the case $F = \{f_1, f_2\}$.

Definition 9 Let $R = (Pl, T, M_0)$ be a Petri net. A non-empty subset F of T is *pre-fusable* with $h \in T$ if and only if there exists a place $p \in Pl$ such that the following conditions are satisfied:

1. The only output of h is p .
($post(h) = \{p\}$)

2. Place p is not an input of h .
($p \notin pre(h)$)
3. Transition h has at least one input.
($|pre(h)| > 0$)
4. Every transition of F has p in its input.
($\forall f \in F, p \in pre(f)$)
5. No transition of F has p in its output.
($\forall f \in F, p \notin post(f)$)
6. Except for h and the transitions belonging to F , no other transition is connected to p .
($\forall t \notin (\{h\} \cup F), p \notin pre(t) \wedge p \notin post(t)$)
7. Place p holds no token initially.
($p \notin M_0$)
8. Transition h does not share its input.
($\forall q \in pre(h), \forall t \neq h, q \notin pre(t)$)

Whenever the above conditions hold, the Petri net can be modified according to the following definition:

Definition 10 Let $R = (Pl, T, M_0)$ be a Petri net, and let $F \subseteq T$, $h \in T$ and $p \in Pl$ satisfy the conditions of Definition 9. The system resulting by the pre-fusion of h and F is $R' = (Pl', T', M_0)$, with:

1. $Pl' = Pl - \{p\}$
2. $T' = (T - \{h\} - F) \cup F'$ with F' defined as:

$$\{hf_i : f_i \in F \text{ and}$$

$$pre(hf_i) = (pre(f_i) - \{p\}) \cup pre(h)$$

$$post(hf_i) = post(f_i)\}$$

where hf_i denotes the concatenation of h with f_i .

Definition 10 is illustrated in Figure 2(a) for the case $F = \{f\}$ and in Figure 2(b) for the case $F = \{f_1, f_2\}$.

The last category of reductions, namely *elimination of redundant places*, is introduced below:

Definition 11 Let $R = (Pl, T, M_0)$ be a Petri net. A place $p \in Pl$ is called *redundant* if and only if there exist transitions t_0, \dots, t_n and places p_0, \dots, p_{n-1} such that the following conditions are satisfied:

1. The only input of p is t_0 .
($pre(p) = \{t_0\}$)
2. The only output of p is t_n .
($post(p) = \{t_n\}$)
3. The only input of each p_i is t_i .
($pre(p_i) = \{t_i\}, i = 0, \dots, n-1$)
4. The only output of each p_i is t_{i+1} .
($post(p_i) = \{t_{i+1}\}, i = 0, \dots, n-1$)

Definition 12 Let $R = (Pl, T, M_0)$ be a Petri net, and let $p, p_0, \dots, p_{n-1} \in Pl$ and $t_0, \dots, t_n \in T$ satisfy the conditions of Definition 11. Then, the net resulting from the elimination of p is $R' = (Pl', T', M_0)$, with:

1. $Pl' = Pl - \{p\}$
2. $T' = (T - \{t_0, t_n\}) \cup \{t'_0, t'_n\}$ where

$$t'_0 = (pre(t_0), act(t_0), post(t_0) - \{p\})$$

$$t'_n = (pre(t_n) - \{p\}, act(t_n), post(t_n))$$

Figures 3(a) and 3(b) illustrate the above definitions for $n = 1$ and $n = 2$ correspondingly. The above transformations preserve the safeness and deadlock freedom of a Petri net⁵. Formally:

Theorem 3 Let $R = (Pl, T, M_0)$ be a Petri net and $R' = (Pl', T', M_0)$ be the net resulting from a sequence of the above transformations. Then, R' is deadlock-free (safe) if and only if R is deadlock-free (safe).

4. Theoretical Extensions

In this section, we extend the theory of reductions in order to get an algorithm for detecting static deadlocks in Ada programs. More specifically, we consider *sequences* of reductions, not just single reductions. As we aim at detecting *all* the deadlocks of a program, we need a stronger version of Theorem 3 which will ensure that no deadlock is lost or added during the reductions. On the other hand, we do not just need to detect that a deadlock exists: we are interested in finding out

what sequences of transitions (or steps in the program) have led to deadlock. This is very important for the designer of a system, because it can help him identify the flaws in his design and correct them. In the following we formalize these ideas. Let $R_0 = (P_0, T_0, M_0)$ denote the initial Ada net and $R_i = (P_i, T_i, M_i)$ be the resulting net after a sequence of i reductions on R_0 . In the following, we assume that R_0 is a safe Petri net, and by Theorem 3, R_i is also safe. The detailed proofs of our results are not given here⁷.

We first examine the case of post-fusions of transitions. The intuition behind the following theorem is that whenever an Ada net is reduced using only the post-fusion rule, and we know a sequence of transitions that leads to deadlock in the reduced net, it is guaranteed that this same sequence leads to deadlock in the initial net. Formally:

Theorem 4 Let R_i be the Petri net resulting from R_0 after a sequence of post-fusions of transitions and let σ be a sequence of transitions leading R_i to a deadlock marking M' . Then σ also leads R_0 to a deadlock marking M , and $M = M'$.

However things are not as straightforward when the pre-fusion rule is used. Intuitively, in the case of pre-fusion, a sequence of transitions that leads to deadlock in the reduced Ada net, does not always lead to deadlock in the initial one. However, it leads 'very close' to deadlock. This means that after applying this sequence to the initial net, only a bounded number of further transition executions may occur. Formally:

Theorem 5 Let R_1 be the Petri net resulting from R_0 after a pre-fusion of transitions F with transition h and let σ be a sequence of transitions leading R_1 to a deadlock marking M' . Then, either σ leads R_0 to the same deadlock marking M' , or σh leads R_0 to the deadlock marking $(M' - pre(h)) \sqcup \{p\}$.

Theorem 6 Let R_i be the Petri net resulting from R_0 after a sequence of pre-fusions of transitions, and let σ be a sequence of transitions leading R_i to deadlock. Then, $\sigma u_{i-1} u_{i-2} \dots u_0$ leads

R_0 to deadlock, where $u_m, m = 0, \dots, i-1$, is either equal to h_m or empty.

The above theorem suggests that if the h_m 's are stored during the reductions, one can later restore the sequences that lead to deadlock in R_0 , by properly extending the sequences that lead to deadlock in R_i .

Finally, if a sequence of redundant place eliminations is applied on the initial Ada net, then the deadlock sequences of the reduced net also lead to deadlock in the initial one. Formally:

Theorem 7 Let R_i be the Petri net resulting from R_0 after a sequence of eliminations of redundant places. Then if σ leads R_i to deadlock, it also leads R_0 to deadlock.

All the results presented until now, can be used to construct an algorithm for efficient deadlock detection in Ada programs. However, we need one more result, which ensures that no deadlock is lost during the reduction procedure:

Theorem 8 Let R_i be the Petri net resulting from R_0 after a sequence of reductions. Let δ_0 and δ_i be the number of deadlock markings in R_0 and R_i correspondingly. Then, $\delta_0 = \delta_i$.

5. The Algorithm

In this section, the proposed approach for deadlock detection in Ada programs is presented. An informal description of the algorithm is given below:

The input of the algorithm – an Ada program – is initially transformed into its corresponding Ada net. In order to reduce the number of states that have to be searched for deadlock, a sequence of transformations is applied on the net. Initially, redundant places are removed. From Figure 3, it is obvious that the elimination of redundant places reduces in general the number of inputs and outputs of transitions. This fact increases the probability that the new net will contain pre-fusable or post-fusable transitions. The net is then searched for fusible transitions. When the final irreducible net is obtained, its reachability

graph is computed, the possible deadlocks are identified and paths leading to those deadlocks are detected. However, this is not enough. We are interested in the paths that lead to deadlock in the initial Ada net, not the reduced one. Such information would allow the designer of a system to identify the deadlocks, and modify the system in order to avoid them. Thus, we use the theory that was developed in the previous section to extend the paths that have been found, getting in this way the paths that lead to deadlock in the initial Ada net. The algorithm is described below:

Input: An Ada program P .

Output: Paths leading to deadlocks in the corresponding Ada net.

Transform P into the corresponding Ada net $R_0 = (Pl_0, T_0, M_{00})$;

Eliminate Redundant Places;

WHILE \exists post-fusible F_m and h_m **DO**

 Apply post-fusion rule;

END;

Initialize H to empty;

WHILE \exists pre-fusible F_m and h_m **DO**

 Apply pre-fusion rule;

$H := h_m \circ H$;

END;

Let H be $h_{i-1} \circ h_{i-2} \circ \dots \circ h_0$, where i is the number of pre-fusions of transitions that have occurred.

Obtain the reachability graph of the reduced Petri net;

For each sink node n of the graph, find a path p_n which starts from the root of the graph and leads to n ;

IF pre-fusion rule has not been used **THEN**

 Output the set of paths found;

ELSE

FOR each path p_n **DO**

 Follow the path in the initial net R_0 ;

 Let the path lead to a marking M_n ;

FOR $j = i - 1$ **TO** 0 **DO**

IF h_j can fire in R_0 **THEN**

$p_n := p_n \circ h_j$;

END;

 Output p_n ;

END;

Before illustrating the algorithm by an example, some comments are necessary:

First, the algorithm always terminates. This is due to the fact that each time one of the proposed transformations is applied, the size of the net is reduced: each place elimination removes one place and each fusion of transitions reduces the number of transitions by one and also removes one place. As we have a finite initial net, the reductions terminate in a finite number of steps.

Second, the reductions can be performed very efficiently. In the case of a fusion of transitions, one must consider a place of the net and check if the transitions in its preset and postset satisfy the required conditions. However, the number of transitions in the preset and the postset of a place, is very small in practice, and of course bounded by the number of transitions in the whole net. Therefore, fusions can be performed very efficiently. On the other hand, detecting a redundant place consists in finding a place with only one input transition t_0 and only one output transition t_n , such that t_0 and t_n satisfy the properties of Definition 11. These properties can be easily validated by a depth first search algorithm that starts from t_0 . During the depth first search, all the places visited, are examined to ensure that they have just one input and one output. Clearly, redundant places can be detected efficiently as well.

Concluding, we should point out that the Ada net representation of an Ada program is usually very small compared to the size of the corresponding reachability graph. In this sense, it is worth performing the reductions on the net, than trying to find techniques that would operate on the reachability graph directly.

6. An Example

In the following we give a complete example of the applicability of the proposed algorithm. We use

a producer-consumer program³, which contains a deadlock. The corresponding Ada net is given in Figure 4. The algorithm first locates the redundant places, which are indicated with a shaded circle in Figure 4. After the removal of these places, we have a sequence of 8 post-fusions of transitions and 2 pre-fusions. The pairs of post-fused transitions are:

t_{13}	t_4
t_{12}	$t_{13}t_4$
t_6	t_7
t_6t_7	t_{16}
t_9	t_{10}
t_9t_{10}	t_{11}
t_{18}	t_{19}
$t_{18}t_{19}$	t_{20}

The pairs of pre-fused transitions are:

t_{17}	$t_{18}t_{19}t_{20}$
t_8	$t_9t_{10}t_{11}$

The reduced net is shown in Figure 5. According to the algorithm, the h_m 's are stored during the pre-fusion reductions, and we have $H = t_{17} \circ t_8$. In the reduced net, we represent with $u_i, i = 1, \dots, 4$, composite transitions that result from the fusions. More specifically:

u_1	$=$	$t_{12}t_{13}t_4$
u_2	$=$	$t_6t_7t_{16}$
u_3	$=$	$t_8t_9t_{10}t_{11}$
u_4	$=$	$t_{17}t_{18}t_{19}t_{20}$

The reachability graph of the reduced Petri net is then obtained, a deadlock state is detected, and the path $p_1 = t_1t_5t_{15}u_2t_3$ that leads to this state is reported. Following p_1 in the initial net, we get the marking $\{p_2, p_5, p_8, p_{12}, p_{15}, p_{19}, p_{27}\}$. According to the algorithm, the only two transitions that may fire are t_{17} and t_8 . This is really the case, and after their execution the initial Ada net becomes deadlocked. We should note here that the initial reachability graph had 82 nodes, while the reduced one has only 15 nodes.

7. Conclusions

An efficient algorithm for detecting static deadlocks in Ada programs has been presented. The algorithm uses Petri net reduction techniques in order to reduce the overhead of the state enumeration approach. The reductions used, have the important property of retaining the deadlock information of the initial Ada net.

Clearly, the proposed approach is superior to the state enumeration algorithms^{1,2}. Moreover, it is simpler and has given better results than the approach based on Petri net invariants³. More recent and independent work⁸, also uses Petri net reduction techniques in order to achieve the same goals as ours. However, our approach has a better theoretical basis as it was first used on process algebras⁷ such as Milner's CCS⁹. Moreover, in our approach we propose a specific algorithm for applying the reductions, something that is missing from⁸. Finally, our technique for finding the deadlock marking in the initial Ada net gives the chance to the designer of the system to reconstruct the sequences of transitions that have led to deadlock. This is very important if the goal is not only to detect hidden deadlocks in the design, but also to correct them.

References

- [1] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362-376, Sept. 1983.
- [2] S. M. Shatz and W. K. Cheng. A Petri net framework for automated static analysis of Ada tasking behavior. *The Journal of Systems and Software*, 8:343-359, Dec. 1988.
- [3] B. Shenker T. Murata and S. M. Shatz. Detection of Ada static deadlocks using Petri net invariants. *IEEE Transactions on Software Engineering*, 15(3):314-326, Mar. 1989.
- [4] G. Roucairol G. Berthelot and R. Valk. Reduction of nets and parallel programs. In W. Brauer, editor, *Net Theory and Applications*, volume 84 of *Lecture Notes in Computer*

Science, pages 277–290. New York: Springer-Verlag, 1980.

- [5] G. Berthelot. Checking properties of nets using transformations. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 222 of *Lecture Notes in Computer Science*, pages 19–40. New York: Springer-Verlag, 1986.
- [6] G. Berthelot. Transformations and decompositions of nets. In W. Reisig W. Brauer and G. Rozenberg, editors, *Petri Nets: Central Models and their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 359–376. Springer-Verlag, 1987.
- [7] P. Rondogiannis. Detecting deadlocks in CCS agents using Petri net reduction techniques. Master's thesis, Department of Computer Science, University of Victoria, 1991.
- [8] S. M. Shatz S. Tu and T. Murata. Applying Petri net reduction to support Ada-tasking deadlock detection. In *The 10th International Conference on Distributed Computing systems*, 1990.
- [9] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

Panagiotis Rondogiannis received the Ptilhion in Computer Engineering and Informatics from the University of Patras, Greece, in 1989. He is presently completing his M.Sc. degree in Computer Science, at the University of Victoria, B.C., Canada. His research interests include theory of concurrency and distributed systems. He currently holds a University of Victoria Graduate Fellowship, and has been awarded several Scholarships from the Greek Institute of Fellowships since 1984. His current mailing address is: Department of Computer Science, University of Victoria, Victoria, B.C., Canada, V8W 3P6.

Mantis Cheng received the B.Math. and M.Math. from the University of Waterloo in 1982

and 1984 respectively, and the Ph.D. in Computer Science from the University of Waterloo in 1988. He is currently an assistant professor at the University of Victoria. His research interests include logic programming, operating and real-time systems and theory of concurrency. His current mailing address is: Department of Computer Science, University of Victoria, Victoria, B.C., Canada, V8W 3P6.

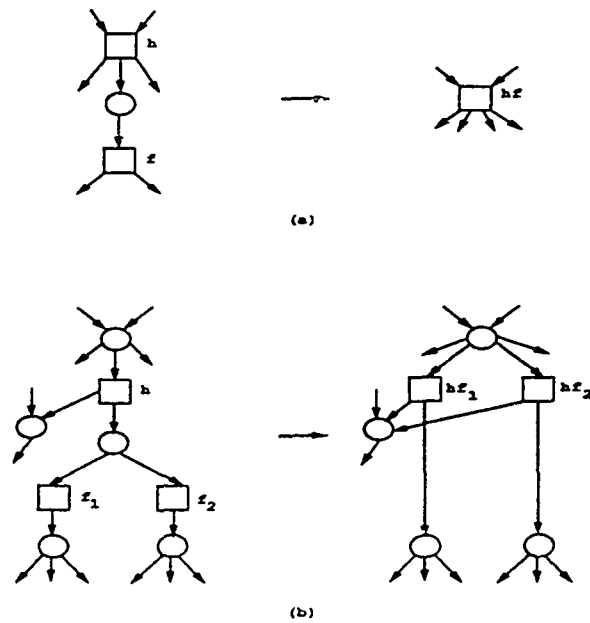


Figure 1: Example of Post-fusable Transitions.

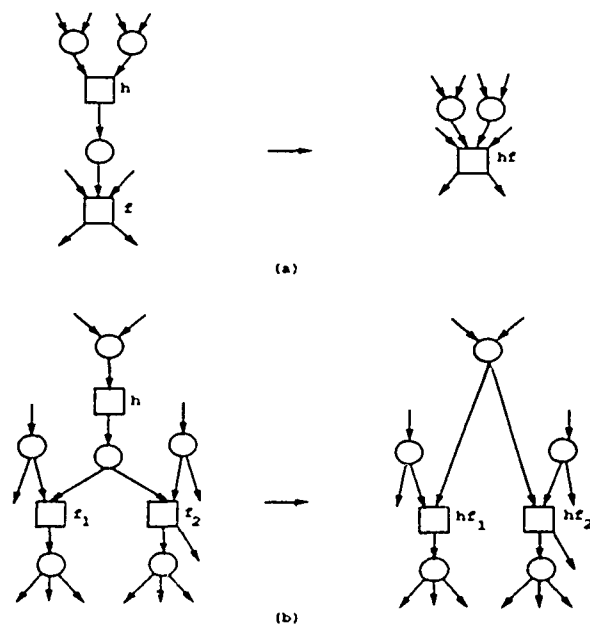


Figure 2: Example of Pre-fusable Transitions.

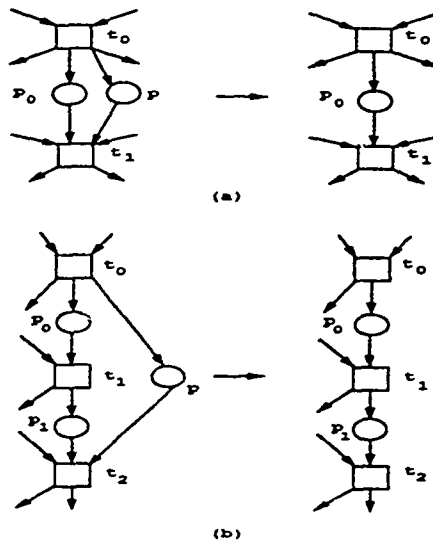


Figure 3: Example of Redundant Places.

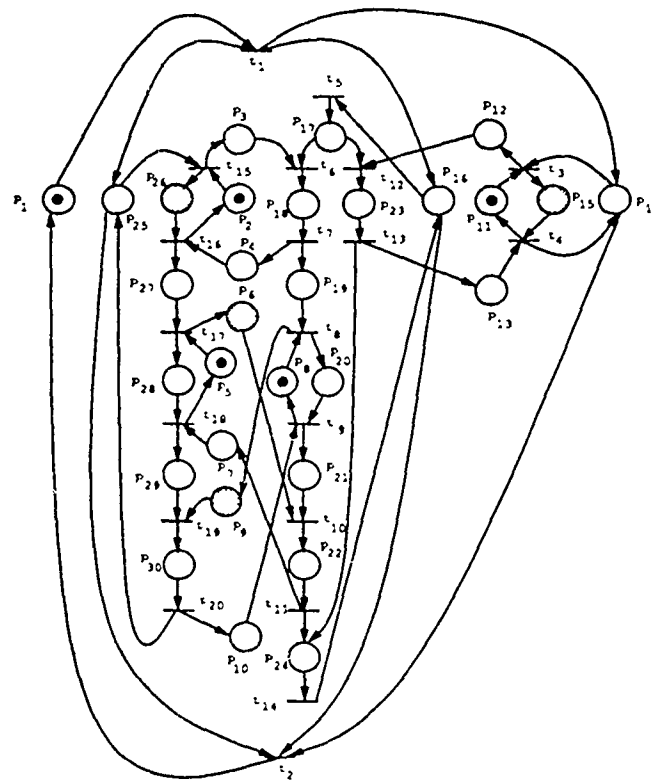


Figure 4: Initial Ada net.

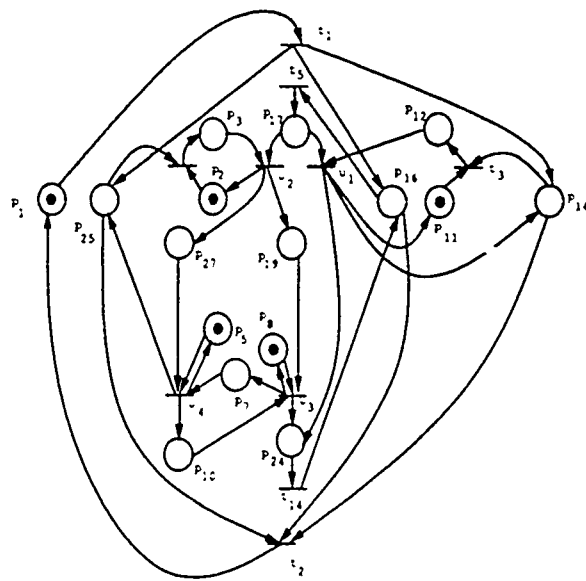


Figure 5: Reduced Ada net.

Ada in the Soviet Union

by Peter Wolcott
Department of MIS
University of Arizona
Tucson, AZ 85721
(602) 626-8682

keywords: Ada, Soviet Union, technology transfer

[abstract] Over the last decade and a half, the Ada programming language has aroused more financial, technical, political, and emotional forces than any other programming language. Because the language is a product of more than just technical factors, its progress is heavily influenced by the surrounding social, economic, and political environment. Judging by the technical characteristics of the language and the goals for its use, one would expect that Ada would be the subject of intense development by the USSR, as well as by the USA. In practice, the Ada experiences in these countries differ considerably. We examine the state of Ada in the Soviet Union from four perspectives: the development of Ada technologies, training and educational efforts, technology transfer within the Soviet Union and from the West, and the levels of interest and support among practitioners, managers, users, and policy makers.

Following the 1979 publication of the Ada Language Requirements Manual, Ada attracted considerable attention in the Soviet Union. A Russian language version was published in 1980, and a high-level Ada working group under the State Committee on Science and Technology was established in 1982 with a mandate to examine the capabilities of Ada, coordinate Ada developments, and determine the need for the language in the Soviet context. By 1987 several tens of Soviet groups were working on implementations. As in the West, the early practitioners seriously underestimated the difficulty of implementing the language. There are today only three to five strong Ada groups and a few dozen less serious projects.

Few organizations have developed full compilers. In the absence of policies mandating uniform implementations and a controlling body analogous to the AJPO, subsets proliferate. The leading work has been done at the Computer Center of Leningrad State University, the Novosibirsk Subsidiary of the Institute of Precision Mechanics and Computer Technology, and the Scientific Research Center for Electronic Computer Technology in Moscow in cooperations with a group of Hungarian organizations. The environments for their compilers typically provide basic sets of tools such as debuggers, linkers, some code management support tools, and testing and diagnostics utilities.

The growth of Ada training facilities corresponds to the growth in activity in compiler development. By 1987, thirty high-education institutions were offering courses in Ada.

Formal and informal transfer of know-how between practitioners has played an important role in the development of the language in the West. While such interaction between Soviet practitioners takes place, it appears to be at a more moderate level than in the West. The Soviet Ada community has had very little direct contact with the West. Although the reforms in the Soviet

Union have made it easier for Soviets to attend western conferences, a lack of necessary hard-currency has proved a major obstacle. Western journals do reach the Soviets, but irregularly, and usually with great delay. The New York University Ada interpreter, and a copy of a DEC Ada compiler have been installed at a number of locations. A few validation suites, from 1984 and 1986 or 1987, have also appeared in the Soviet Union.

A basic characteristic of the Ada experience in the Soviet Union is that, in strong contrast to the US experience, relatively more of the initiative has come from the bottom up. This initiative has not been accompanied by corresponding funding of development, however. A dominant reason is the software development culture of policy makers and managers which places much greater emphasis on efficiency of execution rather than efficiency of development and maintenance. Alternative sources of funding such as venture capital are virtually non-existent.

For a strong Soviet Ada community to develop, there must be strong indigenous demand for, and support of the language, economic incentives which reward developers, adequate tools, and exchange of technology, know-how, and experience within the Ada community and with the West, and a solid educational base. The perestroika-related reforms have created the potential for each of these components to improve. The reforms are requiring that organizations pay much greater attention to life-cycle costs, and new organizational forms such as cooperatives, joint ventures, and small enterprises now allow much higher levels of compensation for work performed. As the borders of the Soviet Union open, it is likely that contacts between the eastern and western Ada communities will increase.

IMPACTS OF CASE TOOLS ON THE WORKING PROCESS OF SYSTEM DEVELOPMENT CREW

Violeta Bulc

241 Hugo Street
San Francisco, Ca 94122
(415) 358-5655 (W)

vbulc%ssf-sys.com.dhl.com.@uunet.uu.net

ABSTRACT

CASE (Computer Aided Software/System Engineering) tools are a subject of contradictory opinions among people involved in development of information systems. Developers of CASE and a majority of information systems gurus state that CASE tools are urgently needed for a successful information systems development. System analysts, designers, and programmers seem to look at CASE more suspiciously. In order to clarify the impact that CASE tools have on the working environment of a "system development crew" analysis of literature and a survey among information systems professionals using CASE tool have been performed.

OVERCOMING THE SYSTEMS DEVELOPMENT CRISIS

Modern corporations are becoming increasingly dependent on data concentrated on a different hardware platforms and organized by diverse database structures. Yet, their ability to organize that data efficiently is reaching the limits of the existing systems and tools available. The consequence is that organizations are losing control over systems development and maintenance. It seems that hardware can not be blamed for such a situation.[2] In fact, computers are becoming faster and more powerful every day. The problem lies very likely in the capacity and complexity of software applications, and in the organization of information systems development.[1,4] According to the research done by Paul Attwell, professor at the State University of New York at Stony Brook,[4] the investment that has been made by corporations in information technology gave almost no increase in productivity. Furthermore, new technology generates more work for the end users and adds an additional layer of administrative overhead to manage new systems. Therefore, to satisfy the

demands for a high quality and performance of the competitive market, the companies need a new approach to systems development. A new technology should be able to overcome the potential difficulties, and ensure streamlined systems development, maintenance and control.

In the response to problems in system development life cycle (SDLC), computer-aided software/system engineering (CASE) tools have been welcome as a possible solution. A CASE tool is defined as "an interlocking set of formal techniques in which enterprise, data, and process models are built up in a comprehensive knowledge base and are used to create and maintain data processing." [6] CASE links data modeling, business modeling, information strategy planning, and code generators. It also encourages high involvement of end users in design and prototyping of the system, as well as the involvement of top management in setting priorities and defining information needs. Yet, the tool's implementation affects the tasks, structures and human subsystems within an organization. CASE requires changes in working habits of employees and their way of thinking.

Strengths Of CASE Tools

CASE implementation has some important strengths. First, it supports system development in the context of the company's overall business strategy and plans. This is accomplished by the storage of system specifications in a central information repository. An information repository documents the business information, their resources, logical models that support those resources, and information about their relationships. Second, CASE facilitates easier system prototyping through the capability to change specifications and determine the impacts of alternative features on the performance of the system. Third, it enforces the use of corporate standards for all phases of the SDLC within the company. Fourth, due to embedded iteration of required system development phases, it enforces easy tracking and upkeep of development steps through the system development life cycle, e.g., easier system control and documentation. Finally, it encourages the use of simplified command interfaces, the elimination of the alien syntax, and the extensive use of graphics screens.

Weaknesses Of Case Tools

The weaknesses of CASE tools are strong enough to have a negative impact on their use. First, many of them rely on a particular system development methodology. Therefore, system developers who used different development methodology before the CASE implementation, may have problems overcoming the changes caused by a new methodology. Second, no standards for CASE tool have been introduced yet. Thus, decision makers have to be careful in selecting a tool. Third, a CASE tool needs skilled developers who can unleash its features and make them functional for the company. The time spent for training may cost three times as much as the tool itself [8] and if the company adds the cost of unproductive time of employees while in a training program, the overall initial cost of CASE tools can be very high. In addition, CASE packages are still facing technical difficulties, e.g., the lack of integration, user-friendly features, and poor presentation capabilities.

Impacts Of The CASE On System Development

While the number of companies buying CASE tools is increasing, the tools equate to a culture shock for system developers. System analysts and programmers, who usually advocate changes, resist the implementation of CASE tools which directly affect their job domain. The implementation of a CASE tool requires the introduction of organizational changes and the re-education of all participants in system development. The users need to understand that a CASE tool will not "think" for them but just help them to be more efficient, faster and better organized. The managers need to realize that it is not enough to choose the tool and calculate the expenses; the human factor must also be considered. Mastering the behavioral problems might be even a greater challenge than the decision about the tool itself.

Case Tools Today

Review of the relevant literature leads to the following conclusions:

1. CASE tools are a new technology that need time to be fully understood. The use of CASE tool is a strategic decision that requires a long term company commitment.
2. CASE tools have a significant impact on the systems development. Developers spent more time in the early stages of the system development life cycle, e.g., preliminary investigation and analysis, and less for programming and maintenance of the systems.
3. CASE implementation affects the tasks, structures, and people within the company. Lack of visible benefits when starting to use CASE causes frustration and dissatisfaction among its users.
5. CASE implementation causes high training expenses which are usually three times higher than the price of the tool itself.

6. Due to lack of experience with and confused expectations about the capabilities of CASE tools, resistance from users of CASE is to be expected.

7. In order for planned changes to be successful, the following key elements are needed:

- well-defined organizational framework
- extensive training
- company-wide commitment
- team interaction
- user's participation
- methods for measuring the productivity and quality of work done by using a CASE tool

THE RESEARCH

The purpose of the research was to gain insights into the work experiences of the "system development crew" using a CASE tool and to recognize the tool's impacts. A variety of techniques could be used for a successful inquiry about the subject, e.g., interviews, questionnaire, study of the existing documentation. To avoid spending too much developers' time and still achieve a reliable level of diagnosis, a questionnaire was chosen as the most suitable one. The questionnaire was designed to be short and simple to encourage a fast and high response rate. It was distributed among 40 system developers and managers at a major company in San Francisco. Twenty one were returned for a response rate 52.5%.

The participants of the survey were identified by the manager of the Information Systems department as representatives of diverse points of view. They were asked to evaluate the following items:

- organizational changes and management support
- communication links
- education
- documentation
- productivity and quality of work
- the CASE features

To understand the results of the survey correctly few additional facts have to be stated:

- the company had at the end of 1990, two hundred and two (202) application developers out of three hundred and seventy (370) total MIS employees
- the CASE tool has been made by a leading CASE tool vendor and evaluated by an independent survey as the best overall fit for the customer requirements with some limited customizability
- most of the developers had only one year of experience in using the CASE tool
- most of the participants never used any other CASE tool
- none of the projects had gone through all of the phases of the SDLC using the CASE tool at the time of the interview

The idea for the elements that were studied in the research came from the schema of organizational relationships and their correlations described by professor Miran Mihelcic, University of Ljubljana, Yugoslavia. [7] Professor Mihelcic defined five basic types of organizational relationships that exist between members of a company: technical, personnel, co-ordinative, communicational, and motivational type of organizational relationships. He showed that in the company business results depend on the quality of these organizational relationships; it also means that if the performance of such relationships is poor the team or a company is very likely to experience some serious business problems. Business related effects were the main reason why the study of CASE impacts on some of the instances of the basic organizational relationships were considered.

Organizational Changes

In its early stages, development of information systems was understood mostly as a programming task. A programmer was responsible for the applications from the beginning to the end, e.g., from its logical design to its implementation. The approach to application development was informal, almost artistic. In the last 10 years information systems professionals have been leaning toward more organized, structured system development techniques. However, convincing system developers to adopt a formal approach to software design is not an easy task. The process of transformation starts with retraining programs for the MIS employees, new definitions of jobs, and formation of new departments. In such a situation, programmers usually complain that they are losing their privacy and that the new jobs are less creative. They think that with CASE everyone's work looks the same. Yet, they are probably aware that through the use of CASE tools the knowledge that used to be a privilege of a few is now available to everyone. The use of CASE requires more effort to be expended in the early stages of the SDLC. This effort pays off in less complex and to some extent easier work for developers later in the life cycle. That means that when CASE tools are employed, software development activities shift from the coding and maintenance (back-end) to the planning, design, and analysis (front-end).

In the survey, twelve respondents agreed that CASE caused organizational changes in their company; seven of them did not think so, two did not answer the question. The implementation of CASE resulted in the establishment of a new information engineering (IE) department and several new jobs. The IE department formalizes process and data modeling, and administration of system development. Among the new jobs that have been created due to the CASE implementation are: information engineering consultant and application architect. New user-liaison support groups were also organized. Six participants stressed that the tool enforced new standards that they believed were necessary for successful development of a large systems. All the respondents except two agree that usage of a CASE tool was a correct management decision, however, its benefits were difficult to recognize in the early stages of system development. All respondents

except one also believed that the CASE tool strategy was supported by upper management, which was actively involved in its use and the training program provided.

A cooperation between management and system developers might be a significant contributor to the success of system development projects. One comment was that the upper managers' "commitment [to CASE] is more verbal than real." The same respondent also added that the management pushed a lot toward fast installation of the systems to meet perceived short-term business needs rather than make a commitment to a long-term process. Some of the respondents doubted that the upper management was aware of the resources and time requirements needed to effectively implement the systems developed using the CASE tool.

Education

The on-going changes in information technology cause problems to people actively involved in information systems (IS) development and maintenance. To be a participant in such a dynamic field, IS professionals are forced to learn new packages on an ongoing basis. These packages are not only new products but, in most cases, they introduce new logic and a unique way of thinking, e.g., relational databases, CASE tools, expert systems, object-oriented languages. A lot of information system professionals cannot handle fast changes without resistance and frustration. However, a well-structured education system may prepare people for the changes, e.g., future directions in information science. Unfortunately, many employees involved in information systems development and maintenance merged into that profession from other areas. The reasons for such a move were usually a better salary and a future with a lot of opportunities. In many cases that means spending several years of studying and using only a particular type of application or system, and any change in the job description may discredit the employee's experience. One of the common reasons for the failure of the CASE are the employees who do not understand the methodology. [8] The employees might know how to handle the tool, but that does not guarantee that the system model will be well defined.

Training. Well-structured training has an important role in a successful implementation and the use of a CASE tool. Organizations have to approach to the training program seriously, matching the course material with the development phase that the system is currently in. Managers need to understand that the initial training usually causes overall confusion, i.e., new terminology, methodology, theoretical approach and fast overview of CASE features. With an insufficient further training program users might not be confident enough to transfer the new methods to the existing processes and models.

In the survey, all except two respondents answered positively to the question of whether they had enough training or not. However, the comments that followed were critical and valuable. One of the dissenting respondents stated that

there had not been enough training because there had not been a need for it up to that time. The other dissenting participant described the training as superficial, concentrating only on the direct use of CASE. The rest of the users agree that the training was well organized and helpful. However, one respondent complained that in spite of organized training additional self-teaching had to be done; such a training was difficult due to late and insufficient manuals. In addition, two of the users mentioned that the training was sometimes offered too early. One user found the scheduling of the training right before it was needed as a very important strategy. Due to the diversity of the participants in the survey, the contradictory answers described above were expected. One noticeable statements came from a user who described the training process for the CASE as a "shallow" mechanical aspect, "the bigger training is learning how to incorporate the goals and concepts into [the CASE tool], which nobody teaches."

Learning Curve. A long learning curve is supposed to be one of the biggest barriers in the use of CASE tools. One of the studies shows that the time spent individually learning a CASE tool even exceeds the time spent in structured courses or seminars. "The learning curve was much more difficult than we thought it would be," said George Schackelford, director of management information systems for SCI. "The training was expensive, the package was expensive and then we found we had to spend several months to become proficient in it". [8] Another study presents that majority of CASE users complain, but they seem to just accept the fact that learning CASE takes a long time and significant effort. The reason for such comments might be a lack of experienced people in the CASE environment. That will probably change when skilled professionals for CASE technology become more common.

Alternatively, all except two participants described the learning process as an easy one, stressing, however, that the manuals were bad and not user-friendly at all. One of the respondents said that the learning curve was about the same as the learning curve for the any other new technology. The one that had some previous experience in using CASE tools added that it was fairly trivial to adapt to the tool if you had data modeling background. One of the managers said that the CASE "automates a common-sense approach to system development." Yet, the user who disagreed with the majority said that as an old main frame user he had to learn PC operations and other software products which made his learning curve much longer. When asked about the duration of the learning period for the CASE tool, 41% of the respondents agreed that it was longer than the training period for other software products, 59% thought that it was the same.

An important conclusion that might be made based on the survey responses is that even though the training is well organized it can miss its purpose if it is not adjusted to the users' current needs. CASE tools require a variety of knowledge, so diverse types of courses are advisable for CASE users. Additionally, quality manuals contribute to

better overall morale and acceptance of a new tool, and need to be available to users in time.

Documentation

Traditionally, the most tedious work in the system development process is the development of system documentation. System developers are asked to produce documentation for the existing system and after that (or at the same time) documentation for the proposed information system. Documentation is important for future system development, i.e., upgrading and maintenance, and facilitates the understanding, evaluation and communication of all system components and people involved in the project. It usually consists of narratives and different diagrams which graphically represent a set of data present in the system, and their relationships. When done manually, documentation is usually poorly maintained and difficult to access by all members of the team. Furthermore, such documentation may contain errors and inconsistencies that might be difficult to detect. CASE tools were made to overcome the problems of the manual documenting.

Almost one hundred percent of the respondents in the survey indicated that project documentation using CASE was better than before, adding that mostly because previously they had not had any. They agree that the CASE tool enforces standardization of system documentation, which makes it easier to read and follow. Other beneficial features mentioned were centralized sources and structures for capturing information in the central repository, which support simplified centralized reporting. One of the respondents pointed out that "there is so much of it [documentation] that it is hard to be effective." However, several technical difficulties were mentioned by respondents. Some of them are: poor searching capabilities of the tool, limitations in editing, lack of copy function, bad CASE tool documentation, bad integration of supported functions, e.g., no carryover between the analysis and design documentation, and poor presentation functions, which means that many documents that need to be presented have to be downloaded and put into an acceptable form by other software packages.

According to other sources, most managers, however, are rewarded for developing software and not for developing documentation. Therefore, the tendency is not to waste too much time on writing system documentation, e.g., to create the minimum possible. Thus, system developers often resent keeping the specifications up to date as they maintain the system. Because of that, there is a fear that specifications will have little to do with a real system.

Communication Lines

Implementation of new techniques in large organizations requires adequate communication mechanisms for diffusing the knowledge and sharing techniques. CASE tools with a central repository physically utilize such an idea. However, the participants' comments in the survey related to the impacts of the CASE tool on communication lines were diverse and crucial. Users were asked to evaluate

three different types of communication: between technical MIS personnel and managers, among technical MIS personnel¹, and between technical personnel and end-users.

Communication Between Technical Employees And Managers. This type of communication line was the subject of several critiques. Only six of the participants agreed that the efficiency of this type of communication was better than before the use of the CASE, one thought that it was worse, and the rest of them that it was the same. When asked about the quality of communication, eight added that it was at least better than before, and the rest of them believed that it was the same. Several respondents believed that the communication would improve in the future. One of the comments said that the communication was worse due to a new nomenclature introduced by the information engineering techniques. Another critique pointed out the lack of user-friendly presentation features of the CASE. Because of that, the information needed for management reports had to be moved from the CASE tool to a drawing program or word-processing packages.

If the CASE documentation itself does not have a practical value for the managers, the managers might not be motivated to use CASE on their own. That is contrary to one of the goals CASE tools want to accomplish: to support managers with current information needed for making decisions, interactively. One respondent added that the reason for the insufficient communication between the developers and managers were not the CASE features but poor managerial skills that could not be overcome just by purchasing a tool.

Communication Among Technical MIS Personnel. The participants agreed that the highest improvement had been made in communication among technical personnel. All except five thought that the communication lines had improved in efficiency and all except two in quality. They agreed that the jobs' boundaries were not so strict anymore. They had to work as an integrated team to perform successfully and effectively.

Communication Between Technical Employees And End-Users. The majority of the participants agreed that CASE improved the relationships between the technical MIS personnel and end-users. The main reason for the improvements the respondents saw in a stronger user involvement in the system development. Two of the respondents said that the communications were worse due to the terminology of CASE, which formed a new level of separation. Another participant added that it was hard for the end-user to understand some of the terms, i.e., business functions, functional decomposition, and mini specifications.

All together, it seems that CASE tools offer a well-defined platform for efficient and improved communication lines among the participants of system development. Technical limitations that vary from tool to tool, and lack of understanding of CASE terminology, however, might be an undesired drawback in the overall development perfor-

mance. The "terminology" problem can be eliminated by the users through the every-day exposure to the use of the tool. The technical pitfalls have to be overcome by the CASE vendors. When evaluating the quality of communication between people there is one concern that might not be so obvious, however, important: a tool can not replace human interactions. People need to communicate and exchange information verbally. In that regard, the comment made by one of the respondent, was applicable, saying that "the quality and productivity of communication lines among people should not be qualified by the evaluation of the tool's impacts." However, a quality tool for storing and retrieving information might be a valuable support for more effective and efficient meetings and conversations.

Productivity

Recent CASE advertisements suggest that users can increase their productivity by 30 - 300%. However, there are few empirical studies done to investigate CASE impacts on productivity. Reasons for that might be that it is, first of all, difficult to measure productivity. Next, the tools for measurement of CASE productivity have become available and reliable very recently. Finally, a lot of companies have no historical productivity results to compare to, e.g., they did not measure productivity before the use of a CASE tool.

One of the rare empirical studies regarding CASE productivity was done by Norman and Nunamaker for Excelerator. [10] The authors believe that the results apply to CASE users in general. Their study shows that through software engineers' perceptions, their productivity improved. The productivity improvements also contributed to a faster development of the system development standards. The authors found that significant since most large enterprises must enforce a rigorous system development methodology and associated standards.

The company where the survey was performed did not use any technique for measuring CASE productivity, at the time of the survey. Thus, the respondents did not have any numeric data available to answer the questions about productivity. Even though, fifteen users believed that CASE would increase productivity, four of them disagreed with that, and two of the user were not sure. The reasons why users believed in increased productivity are the following.

- integrated systems mean less redundant data and functions
- reusable system design saves time for the front-end system development in the future
- CASE keeps the information in an organized structure
- repetitions and duplications are eliminated to the degree possible
- better control and management of information system development

Almost all of the users believed that they would gain some productivity improvement in future. One respondent

stressed the technical pitfalls of CASE, saying that "currently the CASE tool does not work as it should". It was said to be even less efficient than the development tools that were in use before the CASE implementation.

The best productivity evaluation got the project documentation in the information strategic planning and business area analysis phase. Communication lines between the technical MIS personnel and managers got the worst productivity evaluation. An explanation for such an evaluation is that most of the participants have used the CASE tool, so far, only for the information strategic planning and business area analysis phases.

Quality

The quality of work done by CASE tools and the quality of the tools themselves is another area that has been poorly analyzed by CASE users and developers. However, according to Earl Hoskins from AT&T Consumer Communication Service in the U.S., maintenance requests have declined 97% compared to the number of requests that would have been estimated for a project without CASE. [3] His experience illustrated one of the few occasions where a quality metric has been explicitly measured before and after the introduction of CASE. The results are valuable and show the potentials of great savings in the maintenance phase of SDLC, and higher quality of the systems developed by CASE.

In the survey, the highest number of respondents find significant quality improvements in the system documentation. Only a few agree that some improvements have been made in the quality of system control. No argued comments have been stated to support such an opinion. The reason for that might be a misunderstanding of the term "system control" that was not specially defined. In spite of several complaints about the technical characteristics of the tool, 51% of the respondents thought that the quality of the CASE tool itself when compared with other software packages in use was at least better.

CONCLUSION

At this time CASE tools still need a lot of improvements.~ However, the importance and need for CASE tools has been recognized by systems professionals and the tools' features have unleashed forever the forces of computerized software engineering upon MIS professionals and others involved in system development. Probably the most valuable contribution of CASE tools is the availability of large integrated-systems development which was extremely difficult to achieve for large systems before their introduction. System developers have become more organized and effective. Some of the benefits, however, cannot be recognized in the earlier stages of SDLC. In areas such as system documentation, communication lines, and system performance the impacts have been positive in most cases, though, not as high as expected. The CASE has some significant impacts on the developer's behavior. It affects developer's job domain and working habits. As the survey shows, the technical weaknesses of the CASE

cause additional problems. Software must not only work, it has to work well. Procedures must be well defined and documented, which is not characteristic of the present tools.

To overcome the initial problems that might occur in the companies deciding on the use of CASE tools, the managers may consider the following actions as necessary. First, clarify if the tool is really needed. The decision makers may expect more that can be achieved by a CASE tool. Thus, before the managers can make informed decisions about adopting a CASE tool, it is advisable to translate the advantages of that tool into measurable benefits, e.g., to see if the tool meets the company's requirements. Equally important, the disadvantages of the tool must be assessed. Second, choose the vendor carefully. The vendor has to be able to deliver the product on time, adjust the tool to the company's needs, and provide an in-house training. Third, understand the nature of CASE tools. It is important for executives to realize that the real benefit of CASE tool is recognized in the maintenance phase of the system life cycle; for big projects valuable results might not be seen for one, two, or even three years. Consequently training for managers about CASE tools is advisable. Finally, prepare employees for the organizational changes in the company. Some technical skills that were important before the CASE implementation may lose their weight. This can cause confusion and opposition among experienced team members. One way to minimize such reactions is to select a high quality CASE package and provide the employees with well-structured and high-quality training.

To introduce the stated information as a general truth, additional studies with a larger number of participants from diverse working environments will need to be performed. For now, something is clear: CASE tools are here to stay. The onus is on the CASE gurus, managers, and users to overcome the initial barriers and through an organized training make CASE just another valuable engineering tool.

Violeta Bulc is a network analyst with DHL Systems, Inc. in San Mateo, California. The material in this paper resulted from the author's research for her MS thesis. The opinions expressed in this article are those of the author. Her current interests include information technology transfer, and the use of object-oriented application for network analysis. She is actively involved in research of the impact of modern technology on organizational behavior, at University of Ljubljana. Ms. Bulc received a BS from University of Ljubljana, Department of Electro Engineering and Computer Science and an MS at Golden Gate University, School of Management. She is a member of ACM and Data Administration Management Association.

REFERENCES

1. Aranow, Eric. "Is CASE Too Immature for Real Integration?", Software Magazine May 1990. 89+.

2. Henessy, John and David A. Petterson. *Computer Architecture a Quantative Approach*. Morgan Kaufmann Publisher Inc.: San Mateo, 1990.
3. Hewett, Julian. "Where Is CASE Headed?", *Database Programming and Design* October 1990: 41-43.
4. Keyes, Jessica. "Gather a Baseline to Access Case Impact." *Software Magazine* August 1990: 30-43.
5. Martin, James. "Changing Technology Calls for New Information Strategies." *PC Week* 25 June 1990: 73-~
6. Martin, James, and E.A. Hershey III. *Information Engineering: A Management White Paper*. KnowledgeWare 1
7. Mibelcic, Miran. Measurement of an Organization's Dimensions as a Support of Decision-Making Systems for (Governing) and Managing of Companies, paper, Proceeding of International Conference on Organization and Information systems, Bled, Slovenia, Yugoslavia, Sep. 13-15, 1990.
8. Nelson, Ryan R., and Marcus Loh. "Raping CASE Harvest." *Datamation* 1 July 1989: 31-4.
9. Norman, Ronald J., Gail F. Corbitt, Mark C. Butler, and Donna D. McElroy. "CASE Technology Transfer: a Case Study of Unsuccessful Change." *Journal of System Management* May 1989: 33-8.
10. Norman, Ronald J., and Jay F. Nunamaker, Jr. "CASE Productivity Perception of Software Engineering Professionals", *Communication of the ACM* September 1989: 1102-8.
11. Palmer, John F. "The Good, the Bad, and the Ugly." *Database Programming and Design* October 1990: 30-8.
12. QED Information Sciences, Inc. *CASE The Potentials and the Pitfalls*. Wellesley: I. Chantico, 1989.

DON'T TRASH OLD CODE: RECYCLE, RENEW AND CONVERT IT TO ADA

Joseph M. Scandura, Ph.D.

University of Pennsylvania and

Scandura Intelligent Systems

1249 Greentree, Narberth PA, 19072

Abstract -- This paper reviews the current status of the re-engineering industry and then proposes a new cognitive approach to system maintenance which can both dramatically improve systems and minimize Ada renewal costs. This cognitive approach involves modeling and testing the structural and functional essence of a system at a high level of abstraction, with increasing specificity until contact is made with available data and computational resources. The process is essentially the same whether structural analysis (i.e., the cognitive technology) is used to design and develop new systems or to re-engineer old ones. In the former case, the to-be-developed system exists only in the mind of the analyst, designer and/or end user. In the latter case, one begins with a fully functioning system. In both cases, heavy use is made of reusable routines (with new systems) and/or of code salvaged as a result of re-engineering.

Index Terms -- Design, Re-Engineering, Translation, COBOL-Ada, FORTRAN-Ada, reusability.

Saddled with obsolete but essential COBOL or Fortran, DoD contractors and DoD agencies are faced with a series of unpalatable choices. One option is to simply continue with the same old software, patching it where possible to meet the most pressing needs.

Given recent DoD directives, old systems in COBOL, Fortran and other languages requiring significant change must be rewritten in Ada. Properly done and

implemented, either reengineering or redevelopment can add a measure of efficiency previously impossible -- and cost recovery can often be accomplished over a reasonable period of time. The initial costs involved in such renewal, on the other hand, are often prohibitive.

Faced with this dilemma, what are decision makers to do? In my talk, I will review the current status of the re-engineering industry. I then propose a new cognitive approach to system maintenance which can both dramatically improve systems and minimize Ada renewal costs.

1. RE-ENGINEERING: CURRENT STATUS

Several classes of re-engineering tools have evolved over the past few years which appear to offer an array of choices. Indeed, re-engineering today is one of the "hottest" topics in software engineering. There is a good deal of confusion, however, as to just what re-engineering involves, and even more so as to benefits of the kinds of re-engineering tools that are currently available. In this paper, I shall review analysis, design recapture and system redesign tools, along with the major

strengths and limitations of each type. Several tools are mentioned where appropriate.

Code Analysis -- One class of tools involves the analysis of code. Analysis tools are used to determine the complexity of existing systems, and to provide calling hierarchies, cross reference lists and other information concerning the organization of code (or the lack thereof). Restructuring tools and pretty-printers fall in the same general category. They either gather information about existing systems and/or simply represent that information in better form. In each case, the results are referenced by programmers in modifying code.

Clearly, analysis yields positive benefits. Tools in this category, however, have a major limitation. Information about a system and the corresponding code are essentially separate in analysis tools. One can gain insights or information from the analyses, but programmers still have to find the source of those problems and ways to fix them. Other tools, (e.g., editors) are needed for this purpose.

Design Recapture -- A second, somewhat newer class of tools is concerned with "design recapture" -- analyzing source code to determine and visually represent relationships between source code modules. Typically, the information obtained is represented in some type of structure chart, or module calling hierarchy. The basic technology generally involves simple parsing techniques in which modules are identified and attendant relationships captured for later visual representation. The process is not unlike extracting a table of

contents from a book. That is, one looks for headings and similar kinds of information and extracts that information from the body. It is widely recognized that most MIS/DP expense goes into maintaining existing software. Consequently, an increasing number of vendors have begun to introduce tools designed to recapture calling hierarchies -- typically from old COBOL.

Extracting overall relationships within a system and representing them in a visual environment (where they can more easily be modified) is clearly worth doing. Unfortunately, one cannot rely on overall structure in making changes to code, especially where the existing code is poorly designed. As any programmer knows, the "devil hides in the details." Consequently, some re-engineering tools provide limited access to module code. They may eventually also provide direct access to editing tools with which one can modify such code. This approach, if and when it is actually realized, will still leave the biggest problem -- understanding details in order to know how to modify actual code. The kinds of representations (e.g., "bubble charts") used to represent high level designs do not lend themselves well to this task.

Module Visualization -- Solving this problem requires an interactive, visual environment for representing not only overall relationships, such as structured charts, but the structure of individual code modules themselves. This, in turn, requires better ways to visualize code. The purpose of such visualization is to aid human comprehension both by representing structure visually and by eliminating irrelevant detail.

Action diagrams (e.g., Martin 1988; Scandura 1990) help to organize code structure. They only bracket code, however, leaving the human to distinguish different types of structures and to separate relevant from irrelevant detail. Moreover, they are applicable only to low level code. FLOWforms (Scandura 1987, 1990), on the other hand, provide visualization at both levels -- overall relationships and individual modules. In Scandura Intelligent Systems' re/NuSys Workbench™, existing code is reverse engineered automatically into pseudocode FLOWforms, where the pseudocode can be edited, documented, restructured, customized to support multiple environments and used to regenerate full source code. Analyzing existing source code and representing it visually in this manner requires much more sophisticated parsing techniques than simply recapturing designs (i.e., relationships between modules).

Contextual vs. Separate Windowing -- The ability to represent system information visually at these two quite different levels, switching quickly between them as desired, makes it possible to maintain complete systems in one uniform visual environment. In this context, FLOWforms have an advantage over other notations. Although switching between different types of representations (e.g., modules and overall relationships) is best done in separate windows, the use of separate windows is not always desirable. When working either at different levels of a calling hierarchy, or when working at different abstraction levels within an individual module, separate windows make it difficult to remember which windows (i.e., expansions) go with which elements in other

windows. FLOWforms avoid this problem by allowing "explosion" directly in *context*. Lower level detail is automatically displayed within the element which contains it. Thus, FLOWform rectangles may be expanded without in any way affecting the context above or below that in which they exist. This makes it possible to see more detail without losing the general picture. This is *not* possible using graphic elements, such as boxes or circles, connected by lines. Attempting to open a visual element in this type of representation would simply change the overall scale. Consequently, the original context would quickly extend beyond the bounds of the monitor screen.

Customization -- The ability to switch between overall relationships and module detail, and to make modifications at all levels of abstraction, has obvious advantages as regards maintainability. Nevertheless, no one type of relationship will be sufficient in all cases. Calling hierarchies, for example, usually include references to corresponding parameters. But what about global variables? Or, routines exported from one file or compilation unit to another?

What is needed in this case is a way to determine the kinds of relationships to be captured. It would be desirable if the user could cost-effectively create customized representations of his own. The re/NuSys Workbench™'s checking, simulation and high level design generation capabilities provide one such solution. Reverse engineered modules can be checked interactively to more precisely categorize identifiers and their definitions and/or declarations.

Then, higher order routines can be constructed that operate on lower order FLOWform routines and build new FLOWforms more precisely representing the desired relationships.

Multiple Environments.-- Another type of customization involves the ability to support multiple environments in one set of files. Tools from NETRONCAP and Scandura Intelligent Systems support multiple environments in this sense. The NETRONCAP tool is designed for use with COBOL, whereas Scandura's re/NuSys Workbench™ supports C, Pascal, Ada and Fortran as well as COBOL. In FLOWforms this is accomplished by simply labeling structures which are unique to a given platform or operating system. These labels are referenced during code and/or report generation.

Conversion Between Languages.-- In moving to a new environment (e.g., from MVS to DOS or Unix), it is often desirable to convert from an existing programming language into a more modern one. One approach involves the use of source to source translators. These tools take source code in one language and convert it directly into source code in another. This represents a reasonable approach if no further maintenance on the code is desired. In this context, however, one might reasonably ask: "Why translate the code to begin with?" The purpose of translating from one language to another usually is because the software can be better maintained in the new language than in the old. Consequently, it would be best if the conversion were done in an interactive, visual environment -- for the reasons detailed above. Visual FLOWforms

containing pseudocode in one language, for example, might be converted into pseudocode FLOWforms containing another language. In particular, the re/NuSys Workbench™ supports translations from popular older languages to most newer languages: COBOL or Fortran to Pascal, C, or Ada; Pascal to C or Ada, or C to Ada. From 90 to 99% of the code is converted automatically. Higher level designs are preserved in the process.

System Redesign.-- While each of the above capabilities contributes to overall maintainability, all are based on a common assumption -- namely, that the design of the original system is worth recapturing. This raises the following questions: Given a poorly designed system: "Why would one want to capture the design?" Or, if the code is bad, "Why would one want to translate it?"

Many situations call for creating an entirely new or renewed design. Rather than having to build an entirely new system, however, it is possible in some cases to salvage code from the original source by reverse engineering. To be a candidate for reuse, the code may be either highly specific or relatively comprehensive. In most cases, however, it should be highly modular. Perhaps the major advantage of reusing code from an existing system to build a better system for the same or a similar purpose is that large high level modules can often be reused in implementing renewed designs. Experience suggests that, at a minimum, 50 to 60% of existing code, and usually much more -- to over 99%, is reusable in redesigned systems.

Most front end CASE tools support new design. Some also support simulating display and input screens, largely to insure user satisfaction. Both of these factors (i.e., design and displaying user screens interactively) play an important role in system design or redesign. However, they are not sufficient. Confidence in a new, high level design comes only from testing (and debugging) the underlying logic. Such testing can be done only where both data and process are represented in the design, at the same level of abstraction. The design methodologies commonly used in traditional CASE tools favor either data analysis (e.g., information engineering) or process analysis (e.g., structured analysis). Lacking a balanced approach to data and process, they do not lend themselves to debugging designs.

The inability to test the logic underlying high level system designs prior to implementation is a major limitation. As Scandura (1990) demonstrated, the number of tests required goes up exponentially with complexity if all testing is done after implementation, whereas the number of tests required only goes up additively if testing is done from the top down.

In the example cited, the number of empirical tests required in a rather simple system was on the order of 2^{100} if one waited until complete implementation before testing. On the other hand, only about 300 tests were required where testing was done successively from the highest levels of abstraction.

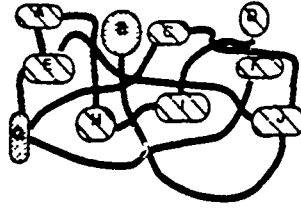
Interfacing Renewed Designs.-- Creating a high level design and testing it, of course, is only one part

of the problem. Another involves the interface between high level designs and reverse engineered, or otherwise reusable code. One solution would be to convert high level designs to the target language and to create an interface between converted designs and the reusable code. To my knowledge, the re/NuSys Workbench™ is the first and, to date, the only CASE and re-engineering solution that explicitly supports the entire process. High level designs are first translated automatically into pseudocode FLOWforms in the target language. (Source code can be generated from such FLOWforms as desired.) In turn, built in checking processes provide an interactive, semi-automatic way to create links between converted designs and the data/process resources referenced in those designs.

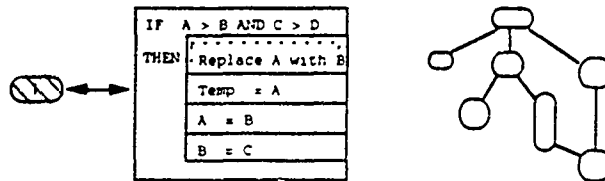
2. COGNITIVE APPROACH TO SYSTEM RENEWAL

This cognitive approach involves modeling and testing the structural and functional essence of a system at a high level of abstraction, with increasing specificity until contact is made with available data and computational resources. The process is essentially the same whether structural analysis (i.e., the cognitive technology) is used to design and develop new systems or to re-engineer old ones. In the former case, the to-be-developed system exists only in the mind of the analyst, designer and/or end user. In the latter case, one begins with a fully functioning system. In both cases, heavy use is made of reusable routines (with new systems) and/or of code salvaged as a result of re-engineering.

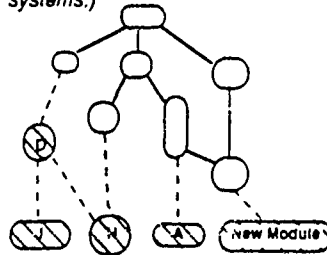
Software Recycling: Renewal via redesign, re-engineering and code reuse



1. "Spaghetti" Code: Most software systems come to look like this as a result of on-going maintenance. Notice that most of the individual modules are fine, and can be reused. The whole system, however, is rather disorganized. (Most new design ideas also start like this.)



2A. Re-engineer Code: Reverse engineer code from the existing system into modular FLOWforms. You easily understand module structure and conveniently make needed repairs. (This step is unnecessary in developing new systems.)



2B. Redesign System: Redesign the desired system at a high level, using PRODOC™'s universal 4GL. Test your design for logical errors, simulating process and data. At this point you should have a hierarchically structured system (Step 2B is optional where the original structure is acceptable.)

3. Reuse Modules: Map reusable modules from your old system into the new high level system design. Finally, use PRODOC™ to design and develop missing low level routines. Experience shows that from 50% to as much as 95% of existing modules are reusable.

In the present context, I shall show how an existing Fortran or COBOL system can be redesigned in a neutral high level design language and converted automatically to Ada, and how the old COBOL and Fortran code can be reverse engineered into a visual environment as pseudocode, restructured where necessary and converted into Ada modules. Finally, I show how the high level Ada design can be linked to the Ada modules.

This cognitive approach is illustrated using the PRODOC re/NuSys Workbench™. I close with a short case history describing application of the approach, and the results obtained therefrom.

Depending on the time available, I plan to demonstrate the above capabilities interactively during my talk.

References:

1. Boehm, B.W. A spiral model of software development enhancement. *IEEE Computer*, 1988, 21, 61-72.
2. Martin, J. and McClure, C., Action diagrams: clearly structured specification, programs and procedures. Englewood Cliffs, NJ: Prentice Hall, 1988.
3. Miller, G. A. The magic number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, 1956, 63, 81-97.
4. Scandura, J. M., Durnin, J. H. and Wulfeck, W. H. III Higher-order rule characterization of heuristics for compass and straight-edge constructions in geometry. *Artificial Intelligence*, 1974, 5, 149-183.
5. Scandura, J. M. Structural (cognitive task) analysis: a method for analyzing content. part 1: background and empirical research. *Journal of Structural Learning* 1982 7 101-114
6. Scandura, J. M. Structural analysis: part 2: toward precision, objectivity and systematization. *Journal of Structural Learning*, 1984, 8, 1-28.
7. Scandura, J. M. A cognitive approach to software development: the PRODOC™ system and associated methodology. *Journal of Pascal, Ada and Modula-2*, 1987, 6 (Sept.-Oct.), 10-25.
8. Scandura, J.M. "Cognitive approach to systems engineering and re-engineering. integrating new designs with old systems," *Software maintenance: research and practice*, 1990, 145-156.
9. Scanlon, D. Cognitive factors in the preference for structured flowcharts. NY: Yourdon Press Authors Conference, 1987.
10. Voorhies D. J. & Scandura, J. M. Determination of memory load in information processing. In J. M. Scandura . Problem solving: a structural/process approach. NY: Academic Press, 1977. pp. 299-316.
11. Yeh, R.T. An alternative paradigm for software evolution. In P.A. Ng & R.T. Yeh (Eds.) *Modern Software Engineering*. NY: Van Nostrand, 1990, pp. 7-22.

THE ECONOMICS OF TRANSLATING SPACE SHUTTLE HAL/S FLIGHT SOFTWARE TO ADA FOR REUSE IN SHUTTLE DERIVED AVIONIC SYSTEMS

Jeffrey E. England
Aerospace Systems Group
Intermetrics, Inc.
Huntington Beach, CA

Roger S. Ritchie
Space Systems Division
Rockwell International
Downey, CA

ABSTRACT

This paper describes a prototype HAL/S-to-Ada translator developed to provide a sound basis for determining the most cost effective approach to developing *Shuttle derived* avionics software in Ada. The implementation of the translator and the economics associated with its use to develop Shuttle derived avionics systems in Ada are described. The Space Shuttle Orbiter Primary and Backup avionics systems are implemented in the HAL/S programming language. Several alternative launch systems have been under consideration by both NASA and the Department of Defense. New avionics software developed for these systems will, most probably, be implemented in Ada. A substantial investment has been made in the development and verification of the mature, highly complex, man-rated Shuttle Orbiter HAL/S flight software. Under some circumstances considerable cost and schedule benefit could be realized by deriving new Ada avionics systems from the mature Shuttle HAL/S flight software through the use of automated translation.

Index Terms -- Ada, avionics, derived, flight software, HAL/S, Orbiter, reuse, software economics, Space Shuttle, translation

1.0 INTRODUCTION

Significant cost of ownership issues and overall international competitiveness have renewed interest in developing alternative launch systems to the Space Shuttle. Innovative and cost effective approaches to developing software for alternative launch systems are becoming increasingly important as their funding will most certainly have to compete with other new start programs such as Space Station Freedom and the Earth Observation System (EOS).

The Space Shuttle Orbiter avionics systems, both Primary and Backup, are implemented in the HAL/S programming language. Several new launch systems have been under consideration by both NASA and the Department of Defense. New avionics software developed for these systems will, most probably, be implemented in Ada. A substantial investment has been made in the development and verification of the operational, mature, multi-path, redundant, highly reliable, man-rated Shuttle Orbiter HAL/S flight software. Under some circumstances considerable cost and schedule benefit could be realized by deriving new Ada avionics systems from the mature Shuttle HAL/S flight software through the use of automated translation.

In the past, automated translation from one high level language to another has not been a particularly popular approach to upgrading large complex systems requiring high reliability. Code generated by automated translators seldom makes use of special features of the target language and is generally inefficient and more complex than the original. Any savings realized by translation may ultimately be offset by higher maintenance costs in systems with long life cycles if the systems are subjected to continuous change. The more dissimilar the source and target languages are, the higher the risk will be of higher overall life cycle cost of the new system. However, because of the potential savings in development and life cycle costs that could accrue through the translation and use of a mature, reliable software system in a stable environment, a project was initiated to examine and develop a translator from HAL/S to Ada.

There is no question that, given a sufficient quantity of the two scarcest of resources, time and money, the technology exists to build a translator to perform a 100% automated translation from any source language to any target language. However, the economic feasibility of building such a translator is a function of many factors: the size of the overall base of software to be translated; the life expectancy of the new software; similarities between the source and target languages; and how closely requirements and interfaces for the new system resemble those of the old system.

Under a research and development contract to Rockwell International, Intermetrics developed a prototype HAL/S-to-Ada translator. The purpose of the project was twofold:

1. Demonstrate the technical and economic feasibility of an automated translation from Shuttle HAL/S to Ada.
2. Provide a sound basis for determining the most cost effective approach to developing *Shuttle derived* avionics software in Ada, whether it be 100% automated translation, 100% manual translation, or some combination of the two.

Three alternative approaches to the translator were examined:

1. Reverse engineering. Convert HAL/S intermediate language, HALMAT, to Ada intermediate language, Diana. Use reverse engineering tools to convert the Diana to Ada. This approach was rejected after determining that insufficient information about the original source code was saved in the HALMAT by the HAL/S compiler. The base of HAL/S software to be translated is too small to warrant modification of the compiler to save the required information.
2. Build an Ada code generator for the HAL/S Compiler. This approach was rejected because the translation would be purely generic (i.e., it would not enable developers to make use of their extensive knowledge of the Shuttle flight software to create a more "Ada-like" derivation.
3. Build the translator from the ground up. This approach was selected because it could be implemented in an incremental manner, a prototype could be created relatively quickly and inexpensively with the aid of Commercial-Off-The-Shelf (COTS) tools, and it would ultimately enable the developers to take full advantage of their special knowledge of the Shuttle flight software to *tailor* the output.

The project was performed in two phases. During phase one a study was conducted to survey the current state of translator technology, identify tools and techniques to be used, measurements to be made, and pitfalls to be avoided during implementation of a prototype. In phase two the prototype was implemented and the results were assessed.

2.0 TRANSLATOR SURVEY RESULTS

Although it was already known that there were no HAL/S to Ada translators in existence, the survey was performed to gather information necessary to establish reasonable expectations for what could be accomplished, pitfalls to be avoided, and to identify COTS translation aids. The translator survey was not exhaustive, but it was fairly extensive. Reference sources included Rockwell's Technical Information Center, the Intermetrics Technical Library, and the libraries of the California State University at Long Beach and the University of California at Irvine.

The results were pretty much as expected, although there were a few surprises. There were no off-the-shelf HAL/S to Ada translators but there are many translation aids available. Two of these, a lexical scanner generator and a parser generator, were used in the development of the prototype translator. Several Ada syntax directed tools were also identified that would be very useful in *manually* translating HAL/S to Ada.

The single most important issue identified was the requirement to perform a thorough source-to-source comparison prior to making the decision to build or not to build a translator. This analysis resulted in the determination that Jovial J73 and Ada have such a wide variety of incompatibilities that a recommendation was made not to build a J73 to Ada translator².

All referenced authors agree that the most significant long-term cost likely to be incurred is reduced readability and maintainability of the resulting code. Typical experience is that, even where the performance of the generated code is acceptable, the code produced is not sufficiently idiomatic in its use of the target language to make subsequent maintenance of the converted software economically feasible. Conversely, Martin⁶ concluded that, "If maintainability and transportability are the goals of the conversion, performance may be reduced." Yellin⁹ suggests preserving program structure during translation to preserve the designer's computational model and, thereby, aid the maintenance programmer in understanding design decisions.

3.0 SHUTTLE HAL/S FLIGHT SOFTWARE TRANSLATION ISSUES

The primary objective was not to build a translator, but rather to determine the most cost effective approach to converting HAL/S applications to Ada, whether it be 100% automated translation, 100% manual translation, or some combination of the two.

A *generic* translator knows nothing about the specific application being translated. Typically, when constructing a translator the objective is to build a generic translator that performs a 100% translation of any syntactically and semantically correct source program. In the instance of Shuttle HAL/S, however, a 100% translation is neither economically feasible nor desirable. The base of HAL/S applications,

compared to languages such as Jovial, FORTRAN, COBOL, or Pascal is relatively small. The subset of HAL/S flight software that is a candidate for conversion to Ada for reuse in an avionic system for a Shuttle Derived Vehicle (SDV) is even smaller. Furthermore, in-depth knowledge of the Shuttle HAL/S flight software, the HAL/S software development environment, and supporting Shuttle flight software data bases enables a more *intelligent* translation than is possible with a purely *generic* language translator.

When describing a compiler, the term "production quality" not only refers to the quality of the generated code, but also includes attributes of the compiler itself, such as speed, ease of use, quality of the documentation, and efficient use of resources. Because the end product would be the translated code rather than the translator itself, the translator did not have to be production quality. The aesthetics of the translation process (i.e., translation speed, resources used, user interface, etc.) were not important. Correctness, efficiency and maintainability of the translated code are the important issues.

3.1 HAL/S versus Ada

HAL/S is a real-time, block structured, procedure oriented, high-order algorithmic language. Many of the features of Ada exist, in some form or another, in HAL/S. Most basic statements can be translated directly. Most HAL/S data types have direct Ada equivalents, although Ada often requires a type name in cases where HAL/S only requires a definition.

Many other features of the Ada language, such as dynamic memory allocation, data abstraction and overloading, do not exist in HAL/S. However, the major concern is HAL/S features that do not exist in Ada. Fortunately, the flexibility of the Ada language facilitates implementation of most of the missing HAL/S features. Some of the more specialized features of HAL/S, however, such as synchronous scheduling and implicit type conversion, are difficult to express with the generalized features provided by the Ada language.

3.2 Reusable HAL/S Software Size

The HAL/S language was developed by Intermetrics under a contract to NASA that began in 1970. It has been in regular use by NASA and NASA contractors since 1973. Eighteen different HAL/S

compilers have been developed to support a variety of hosts and targets. Major applications that have been developed in HAL/S include:

- 1) Shuttle Primary Avionic Software System (PASS)
- 2) Shuttle Backup Flight System (BFS)
- 3) Parts of the PASS and BFS Ground Support Software
- 4) Onboard software for the European Space Lab
- 5) Attitude Articulation and Control System (AACS) for the Galileo Jupiter probe
- 6) Parts of the onboard control software for the Magellan Venus radar mapper
- 7) Parts of the Deep Space Network (DSN)

In addition, many thousands of lines of test cases have been developed for these applications and for the HAL/S compiler itself. Nevertheless, compared to other languages such as Jovial, Pascal, FORTRAN, COBOL, and Ada the total base of HAL/S software is quite small.

The base of HAL/S software that would be reusable for a Shuttle derived avionic system is even smaller as shown in Figure 1. The baseline would probably be the PASS flight software. It is comprised of a total of 215K Source Lines Of Code (SLOCs), of which 178K SLOCs is HAL/S. Without knowing the specific requirements for the SDV it is difficult to estimate that portion of the software that would actually be candidate for translation and reuse. If the SDV is unmanned all the Crew Interface (CI) software can be eliminated. If it is an expendable launch vehicle then such functions as deorbit and descent or any of the abort modes such as Return To Launch Site (RTLS), Trans Atlantic Landing (TAL) abort, or Abort To Orbit (ATO) will not be needed. However, in a manned system such as a second generation Shuttle, or even a Personnel Launch System (PLS), most functions should have application.

3.3 HAL/S Software Profile

Since the base of software that is candidate for translation is relatively small, the cost of building the translator must be quite small in order for the total life cycle cost of the end product (i.e., the SDV flight software) to be cost effective. Researchers already knew from experience with the two languages that some HAL/S constructs can be directly translated into one or more equivalent Ada constructs, and that others have no Ada equivalent. It is also known from

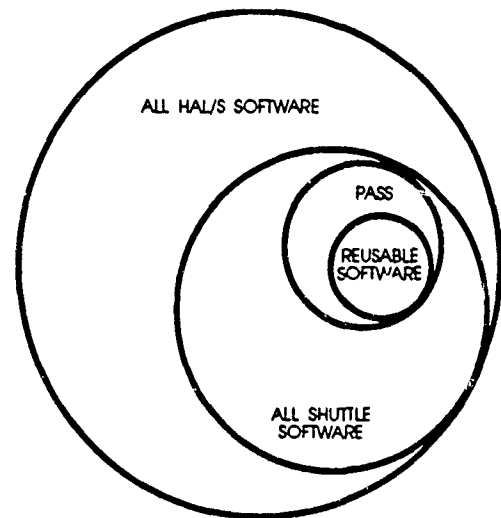


FIGURE 1. HAL/S SOFTWARE BASE

experience with the Shuttle flight software that some HAL/S constructs are used more than others, and that some are not used at all. In order to ensure that efforts were concentrated in those areas that would result in the highest percentage of translation, a comprehensive profile of the software to be translated was generated.

By using a recently baselined Operational Increment (OI) of the Shuttle PASS HAL/S flight software (OI-8C), it was a simple matter to determine the exact subset of the language that is actually used. An automated tool was developed to count occurrences of each HAL/S keyword, built-in function, and statement type. A complete "profile" was performed of that portion of the Shuttle HAL/S flight software that researchers considered the most probable candidate for reuse in an SDV. As suspected, many features of the HAL/S language are not used at all; many others are used infrequently. A summary of the HAL/S keyword and built-in function profile is presented in Table 1. In the interest of brevity the statement profile is not shown.

4.0 TRANSLATOR ARCHITECTURE

A translator is basically a compiler. It converts code from a source language to a target language. The target is generally a lower level language than the source. In most cases it is assembler or machine code native to a specific processor. In other cases, the target may be a processor independent inter-

TABLE 1 HAL/S KEYWORD AND BUILT-IN FUNCTION PROFILE

ABS	512	MAX	13
ABVAL	144	MODAL	551
ACCESS	0	MIN	4
AFTER	0	MOD	23
ALIGNED	0	NAME	13,547
AND	3,226	NEXTIME	0
ARCCOS	27	NONHAL	0
ARCCOSH	0	NOT	2,317
ARCSIN	28	NULL	838
ARCSINH	0	OCT	11
ARCTAN	12	ODD	6
ARCTANH	0	OFF	7,948
ARCTAN2	70	ON	6,137
ARRAY	4,256	OR	3,826
ASSIGN	617	PAGE	0
AT	237	PRIQ	0
AUTOMATIC	106	PRIORITY	161
B&	41,463	PROCEDURE	861
BIT	5,450	PROO	0
BOOLEAN	560	PROGRAM	170
BY	7,254	RANDOM	0
CALL	2,370	RANDOMG	0
CANCEL	58	READ	0
CASE	281	READALL	0
CAT	374	REENTRANT	57
CEILING	4	REMAINDER	27
CHAR	14	REPEAT	60
CHARACTER	507	REPLACE	7,330
CLOCKTIME	24	RESET	80
CLOSE	1,516	RETURN	10
COLUMN	0	REMOTE	81
COMPOOL	467	RIGID	1,336
CONSTANT	866	RJUST	0
COS	133	ROUND	12
COSH	0	RUNTIME	7
DATE	0	SCALAR	5,152
DEC	24,272	SCHEDULE	161
DECLARE	24,263	SEND	12
DENSE	185	SET	105
DEPENDENT	6	SHL	133
DET	0	SHR	81
DIV	30	SIGN	132
DO	13,517	SIGNAL	5
DOUBLE	1,587	SIGNUM	16
ELSE	4,677	SIN	141
END	13,517	SINH	0
EQUATE	463	SINGLE	1,077
ERRGRP	0	SIZE	8
ERRNUM	0	SKIP	0
ERROR	16	SORT	142
EVENT	77	STATIC	0
EVERY	80	STRUCTURE	1,152
EXCLUSIVE	15	-STRUCTURE	10,073
EXIT	67	SUEBIT	961
EXP	31	SUM	0
EXTERNAL	403	SYSTEMA	0
FALSE	474	TAB	0
FILE	0	TAN	22
FLOOR	148	TANH	0
FOR	1,231	TASK	0
FUNCTION	8	TEMPORARY	2,434
GO	0	TERMINATE	0
HEX	18,560	THEN	10,493
IF	10,493	TO	3,913
IGNORE	0	TRACE	2
IN	3	TRANPOSE	22
INDEX	1	TRIM	0
INITIAL	21,854	TRUE	447
INTEGER	7,636	TRUNCATE	12
INVERSE	0	UNIT	123
LATCHED	75	UNTIL	87
LENGTH	0	UPDATE	14
LINE	0	VECTOR	1,045
LJUST	0	WAIT	182
LOCK	0	WHILE	99
LOG	10	WRITE	0
MATRIX	319	XOR	236

mediate language, such as "P Code." Intermediate code is often cryptic and difficult to read and is rarely viewed by an application developer.

The source language is HAL/S. The target language is Ada. Although the target may be viewed as an intermediate language in the overall scope of building a Shuttle derived avionic system, it is the Ada code that must be modified and maintained for years to come by the applications developers. Therefore, in addition to correctness and efficiency, maintainability of the Ada code is also paramount.

Keeping in mind that the translator itself need not be "production quality," as many COTS tools as possible were used to keep development costs of the prototype translator down. Although the prototype would only perform a partial translation, most of the components of the translator had to be fully functional.

The translator was implemented on a Sun Workstation under UNIX. An overview of the translator is presented in Figure 2. Its three basic components are a lexical analyzer, a parser, and a set of translation routines. Although it is not really part of the translator, there is also a fairly large library of support packages to provide an Ada equivalent of the many functions built directly into the HAL/S language. Finally, because the speed of the translator was relatively unimportant, a preprocessor was used to prepare the HAL/S input, and a postprocessor was used to "pretty print" the Ada output after translation rather than build these capabilities into the translator itself. Each of these components is discussed in more detail below.

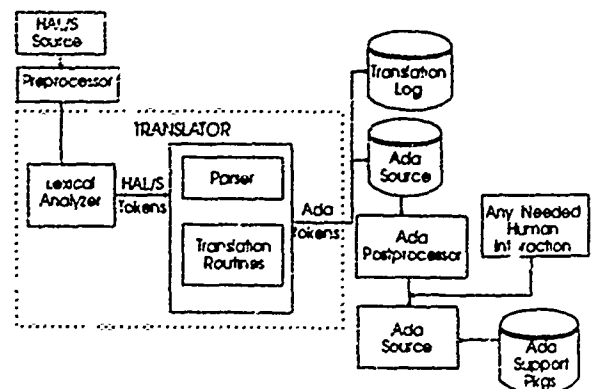


FIGURE 2 TRANSLATION PROCESS OVERVIEW

4.1 Lexical Analyzer

A lexical analyzer partitions an input stream of text into tokens that match expressions supplied by the user in the form of an input specification. A token is a basic element of the language such as a keyword, an identifier, or punctuation. A standard UNIX tool, LEX, was used to generate the lexical scanner for the translator. An overview of this process is presented in Figure 3.

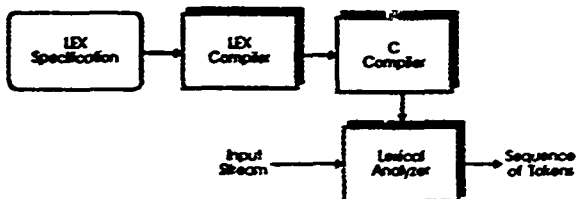


FIGURE 3. CREATING A LEXICAL ANALYZER

The LEX compiler uses an input specification provided by the user and generates C source code. This C source code is subsequently compiled and linked into an executable lexical analyzer that corresponds to the user's original input specification. The input specification for the lexical analyzer was created by studying the HAL/S grammar to determine what lexical elements were needed.

4.2 Parser

A parser organizes its input according to some specified set of grammar rules. The parser for the translator was also generated with the aid of a standard UNIX tool, YACC (Yet Another Compiler Compiler). An overview of this process is presented in Figure 4.

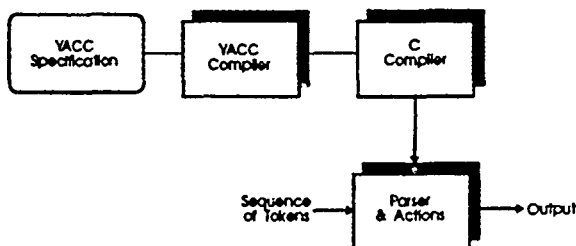


FIGURE 4. CREATING A PARSER

YACC accepts as input a modified Backus Naur Form (BNF) definition of the grammar rules that define the structure of the language to be parsed. The BNF definition of the HAL/S grammar used for

the translator was taken directly from the HAL/S compiler for the Shuttle Backup Flight System. The YACC compiler converts the input specification into C source code that is subsequently compiled and linked with other components to create the executable translator.

The parser calls the lexical scanner to provide input in the form of tokens. These tokens are organized according to the rules specified in the input grammar definition. When a rule is recognized, a program fragment for that rule is executed. These program fragments are called translation routines.

4.3 Translation Routines

A translation routine performs the specific translation of a HAL/S construct into an Ada construct. A translation template is a specification of what a translation routine is to do. Since the translator is a prototype, translation templates were only defined for a subset of the HAL/S language. Using the keyword and statement profile shown in Table 1, an initial subset of the HAL/S language was identified for which translation templates would be defined. A detailed comparison of the two languages was then performed. A translation template consisting of a semantically equivalent Ada statement or construct was defined for each HAL/S statement or construct. The intent was to define the best translation of the existing code, not to redesign it. The structure and design of the original HAL/S code was maintained. Consequently, if the original HAL/S code is not well designed, the translated Ada code will not be well designed.

These translation templates became the heart of the software requirements specification for the translation routines. Two example templates are shown below. Example 1 presents a trivial translation involving no semantic issues: an *IF* statement both with and without an *ELSE* clause.

Example 1

C Translation Template

C HAL/S IF statement with no ELSE clause

```
IF <condition> THEN
    <statement>;
```

C HAL/S IF statement with ELSE clause

```
IF <condition> THEN
```

```

<statement>;
ELSE
<statement>;

```

- Ada Equivalent of
- IF statement with no ELSE clause

```

if <condition> then
  <statement>;
end if;

```

- IF statement with ELSE clause

```

if <condition> then
  <statement>;
else
  <statement>;
end if;

```

Example 2 presents the more difficult translation of local variables. HAL/S local variables are static; they retain their value between calls. Initialization of local variables happens only on the first call. Conversely, Ada local variables are dynamic; they do not retain their value between calls. Initialization occurs each time a program unit is called. In order to implement the Ada equivalent of HAL/S local variables, their declarations must be moved to local packages.

Example 2

C Translation Template
C HAL/S Local Variables

```

OUTER:
PROCEDURE;
  DECLARE IVAR INTEGER INITIAL (0);
  INNER:
  PROCEDURE;
    DECLARE SVAR SCALAR;
    ...
  CLOSE INNER;
  ...
CLOSE OUTER;

```

- Ada Equivalent of
- HAL/S Local Variables

```

package OUTER_VARS is
  IVAR : INTEGER := 0;
package INNER_VARS is
  SVAR : FLOAT_S;

```

```

end INNER_VARS;
end OUTER_VARS;

```

```

with OUTER_VARS;
procedure OUTER is
  use OUTER_VARS;
  procedure INNER is
    use INNER_VARS;
  begin
    ...
  end INNER;
begin
  ...
end OUTER;

```

A total of 355 translation templates was identified during development of the prototype. Researchers estimate this represents approximately 75% of the templates that would be required for a complete translator. Translation routines were developed for 203, or about 43%, of the of the estimated 473 templates that would be required for the completed translator.

An overview of the assembled translator with the translation routines is presented in Figure 5.

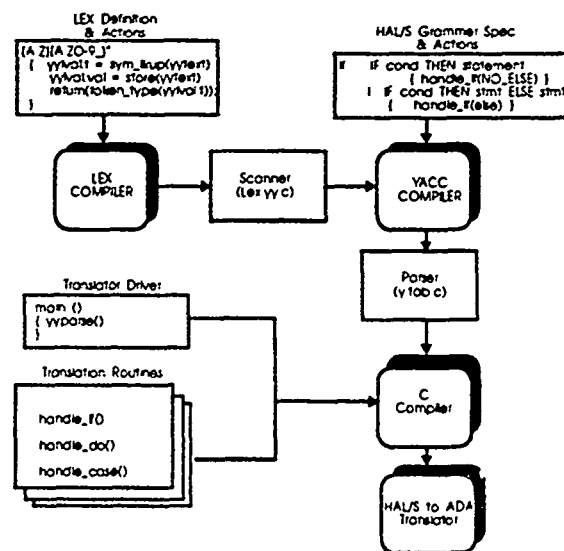


FIGURE 5 PUTTING THE TRANSLATOR TOGETHER

4.4 Support Packages

The HAL/S language was developed for NASA specifically for use in real-time man-rated spacecraft applications. It has many built-in functions and

capabilities incorporated specifically to support such things as mixed mode arithmetic, vectors and matrices, math functions, and quaternions. Equivalent Ada capabilities were provided for the translator in the form of Ada support packages. Some of these packages had already been developed and placed into the public domain by Mr. Allan Klumpp of JPL^{4,5}. Although the packages are available through NASA's COSMIC library, Mr. Klumpp graciously sent copies of them for use with the translator because of the short schedule. The template document provides specifications for other support packages that would be needed for a complete translator. These packages provide for such things as bit string operations, arithmetic operations with arrayed operands, partitioning of arrays, and a variety of type conversion routines.

4.5 HAL/S Preprocessor

The HAL/S compiler performs functions via compiler directives, which are the HAL/S equivalents of Ada Pragmas. They are not executable statements, but rather commands to the compiler that affect how it interprets and uses the information it is processing. Researchers found it simpler to use a preprocessor to process certain of these directives prior to actual translation of the HAL/S source code into Ada.

The REPLACE directive provides HAL/S with a capability similar to the macros of other languages⁶. It instructs the compiler to substitute macro text for subsequent occurrences of a macro name. The preprocessor performs in-line expansion of these macros in the HAL/S source code.

The INCLUDE directive provides HAL/S with a mechanism for executable code to be shared among separate compilation units⁸. Whenever a program, comsub, or compool is compiled, an external block template and Symbol Data File (SDF) is generated that contains all the information necessary to reference that block from within another compilation unit. It is similar to an Ada specification except that it is generated automatically by the compiler. The preprocessor for the translator generates an equivalent symbol file necessary for the translator to perform the INCLUDEs and resolve references to HAL/S objects during the translation.

4.6 Ada Post-Processor

Avionic software developed for any launch system will be maintained for a minimum of 10 years after deployment, probably 30 years or longer. Therefore, maintainability of the resultant Ada code is paramount. Many factors contribute to software maintainability. One of these factors, and probably the most important, is the readability of the code.

The output of the translator is reformatted using an off-the-shelf tool to enhance readability. It is a proprietary version of an Ada listing formatter originally developed by Intermetrics under contract to the Naval Ocean Systems Center (NOSC). It provides substantial latitude and control over the formatting parameters, such as indentation, use of upper and lower case, alignment of colons, use of "white space," and placement of comments. For the purposes of this project, formatting parameters were set according to the style used in the Ada Language Reference Manual⁷. Additionally, the translation templates were designed to produce readable Ada code.

5.0 ECONOMICS OF AUTOMATED TRANSLATION

A key objective of automated translation is to reduce software development costs when compared to the cost of developing the target software from scratch. Another objective is to not exceed the development savings with increased life cycle, particularly maintenance, costs. Consequently, to determine the most cost effective approach to developing a Shuttle derived avionic software system in Ada, it is necessary to examine the total life cycle cost, including maintenance, of each of the alternatives. The three alternative approaches examined were the following:

1. Translate existing HAL/S to Ada and upgrade to new requirements
2. Retain existing HAL/S and upgrade to new requirements
3. Develop the new software in Ada

To determine the total software life cycle cost for alternative 1, automated translation, the following cost elements must be considered:

1. Developing the translator
2. Verification of the translator
3. Operating the translator
4. Manual completion of the translation process
5. Development of new software for the target system
6. Integration and verification of the target software
7. Maintenance of the target software

Each of these cost elements is examined below.

5.1 Development of the Translator

The cost of developing the translator includes both fixed and variable components. Fixed development costs are fairly independent of both the source and target languages. They are comprised of the cost of those components that must be fully implemented, including the pre-processor, lexical analyzer, parser, and post-processor.

The largest cost component is the variable cost of implementing the translation routines. When building a fully automated generic translator, variable cost is driven entirely by the size, complexity, and similarities between the source and target languages. However, in the instance of a HAL/S to Ada translator, a 100% translation is neither required nor economically feasible. Specific translation routines are selectively implemented based on the frequency of occurrence and complexity of each HAL/S construct to be translated. The optimum mix of automated and manual translation is determined through analysis. This analysis takes into account the percentage of automated translation achievable by each individual translation routine and the cost of manually completing the conversion process.

The cost of developing a language translator is represented in Figure 6, Translator Development Cost. For each project, the optimum mix of automated and manual translation will vary. In the case of the HAL/S to Ada translator, using the PASS HAL/S flight software as a baseline for the development of a highly Shuttle derived avionic system, the optimum mix was determined to be automated translation of approximately 75% of the

reusable software base versus 25% manual translation.

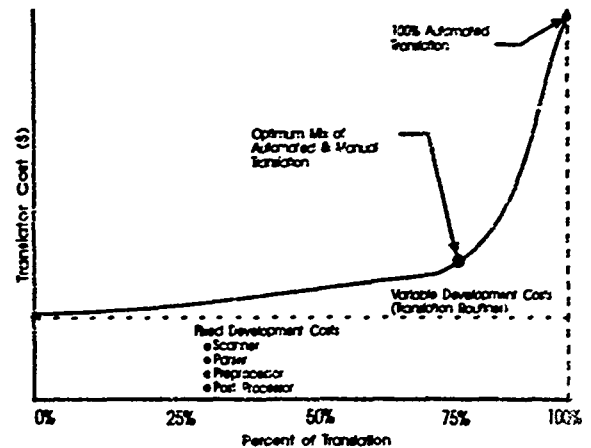


FIGURE 6. TRANSLATOR DEVELOPMENT COSTS

5.2 Tradeoffs Between Manual Conversion And Automated Translation

Several tradeoffs need to be considered when determining whether manual, fully automated, or semi-automated translation is the most economical means of translating the desired software. Some of these tradeoffs are discussed below.

The advantages of automated translation include that it is consistent (i.e., repeatable) for a given input, rapid (once the translator is developed), and non-labor intensive. A major disadvantage is the time and cost required to develop the translator. Additionally automated translation transforms only the code. All other software products, such as documentation and test cases, must still be manually converted.

The advantages of manual translation include that it can be performed quickly and economically on a small amount of code. It also enables restructuring of the code to take advantage of special features of the target language, compiler, or other system features, resulting in more efficient target code. The major disadvantages of manual translation include that it is labor intensive, error prone, generally inconsistent, and less economical than automated translation for large amounts of software.

Both manual and automated translation may be able to reuse, to some extent, high level black-box

integration tests. However, automatically translated software may also be able to reuse *some* lower level black-box and white-box tests. This is because automated translation will result in a higher degree of traceability to source software verification tests than will manual translation. The reason is that the design and structure of the original HAL/S code are maintained, including such things as object names and data types. Manual translation will generally result in these characteristics being altered more than if the software is translated automatically.

5.3 Verification Of The Translator

Verifying the translator is similar to validating a compiler. No amount of testing can ensure that it will function correctly in all situations. The parser and lexical analyzer should not need to be verified if they are created with tools such as YACC and LEX because the process used to generate them is already proven¹. However, the inputs to these tools will need to be verified, probably by peer review.

The translation routines, which are based on the translation templates, are the most crucial component of the translator. Verification of both the templates and the routines will be necessary to ensure that the translation is both syntactically and semantically correct. The translation templates also form a source of ready made tests for the translator. Once a translation template has been verified, it may be also used to create the test cases for the corresponding translation routine. Both valid and erroneous inputs should be tested. Processing of language constructs corresponding to unimplemented translation templates should also be performed to ensure they are properly handled and flagged in the output.

In the case of the HAL/S to Ada translator, the objective is not to build a reusable translator but rather to translate a relatively small base of reusable HAL/S software. The translator itself is essentially a throw-away. It is of no value once the automated portion of the translation process has been completed. Thus, if a translation error is detected for a source software construct that has been used relatively infrequently, it may be more economical to manually correct the error in the target software rather than to correct the translation routine. The erroneous translation routine may simply be modified to flag the location in the target software and print a listing of the statements requiring manual intervention to correct.

5.4 Operating The Translator

The cost of operating the translator is negligible. It is expected to be a one-time, non-recurring cost. Whether it requires an hour, a day, or a week the cost of operating the translator is trivial when compared to the overall cost of developing the completed system. Under some circumstances, however, it may be necessary to run the translator more than once. For example, it may be desirable to translate subsequent releases of the source software to incorporate changes made after the initial translation. It may also be desirable to incorporate optimizations or constraints into the translator for a specific target compiler, computer, or system. It is also possible that the translator may be used to derive different target systems, such as a different SDV or another Space Lab. The decision whether to re-run the translator or to use the target software generated by a previous translation as the starting point for implementing changes must be considered on a case by case basis.

5.5 Manual Completion Of The Translation Process

Manual completion of the translation process will be necessary for the following reasons:

1. Some constructs in the source language may not be directly translatable into the target language (e.g., HAL/S process timing information into Ada tasking)
2. Some constructs are so rarely used in the source software that it is more economical to manually translate their few occurrences than to automate their translation
3. There may be some need to restructure the target software to account for or take advantage of target language features, or to take into account hardware or system differences between the source software system and the target software system, (e.g., I/O, memory structure, etc.)

Manual completion may also include enhancing the overall quality of the target software. Quality enhancements may include such things as the use of

different data structures, control flows, or modularization. The specific mix of automated and manual translation requires a cost-benefit analysis for each target system. A summary of the major software cost trades for a highly Shuttle derived avionic system is presented in Table 2.

TABLE 2. MAJOR SOFTWARE COST TRADES

	Translated to Ada	Ada from Scratch
Avionic Architecture Effect on S/W Costs	Increased H/W changes that are non-transparent decreases S/W translatable (low to high)	Increased H/W changes do not significantly affect outcome (med.)
Requirements Development Costs	More strictly Shuttle derived-lower	Less constrained to Shuttle-higher
Code Development Costs	Lower	Higher
Maintenance	Higher	Lower

5.6 Development Of New Software For The Target System

It is expected that an avionic system for any alternative launch vehicle, even one that is highly Shuttle derived, will differ substantially from the present Shuttle avionic system. The more dissimilar the SDV avionic system is from the Shuttle avionic system, the more new software will have to be developed to complete it.

5.7 Integration And Verification Of The Target (Ada) Software

Verification of a system developed using automated translation will most probably be less difficult than verification of a system developed entirely from scratch due to the possible reuse of source software verification tests. Some additional

sources of errors that need to be accounted for due to translation include:

1. Insufficient run-time efficiency; the translated software will generally be larger, both in terms of SLOC and memory utilization, than the source software;
2. translation errors resulting from semantic differences between the source and target languages; and
3. integration of translated software with software developed from scratch in the target language.

Some of these errors may be mitigated by:

1. Use of a faster processor for the target system than is used for the source system. This is often possible due to advances in processor technology since the source system was deployed.
2. Detailed source-to-source analysis to identify and account for semantic differences between the source and target languages.
3. Use of the verified translation templates to also create translator test cases.
4. Use of loosely coupled external packages to implement features of the source language not directly translatable into the target language, such as built-in functions, mixed mode arithmetic, or unsupported data types.

In the case of the prototype HAL/S to Ada translator, many translator specific verification problems were solved by the above steps. Researchers conservatively estimated that the processor for the target system would result in at least a 50% increase in processing power over the source AP-101/S (1.2 MIPS) processor. The detailed source-to-source analysis involved experts familiar with both Ada and HAL/S. The prototype translator was tested with both translation test cases and a representative subset of the PASS flight software. Utility packages used to implement HAL/S built-in functions, such as math packages, were very loosely coupled with the code generated by the translator, enabling them to be separately verified.

The verification philosophy will depend on the size and criticality of the target system and on the expected frequency of use of the translator. If the system involves a small amount of software to be translated and/or the translator is expected to be used only once, then verification efforts will be focused almost entirely on the translated software. A much smaller effort will be expended on verifying the translator itself. Conversely, if the base of software to be translated is large and/or the translator is expected to be used many times, then a much larger proportion of verification efforts will be directed toward the translator.

Depending on the level of confidence in the translator and maturity of the source software, the verification of the target software may be concentrated on areas of semantic differences between the two languages. A major benefit of translation should be a reduction in overall testing and verification costs due to maturity of source software, confidence in the translator, and possible reuse of source software test procedures.

The use of tools to automatically verify or assist in the verification of the target software should be used whenever possible, such as the Rockwell International proprietary Ada Test Program (ATP). The ATP assists in white-box testing of Ada software. The availability of a particular tool to aid in the verification of specialized features of the target language may be factored into the evaluation of alternative implementations of the translation templates.

5.8 Maintenance Of The Target Software

Maintenance of the target system is a recurring cost and must be carefully analyzed for each alternative implementation. Some of the factors affecting maintenance costs include its maturity, complexity, size, and change traffic.

For software of equal maturity, the maintenance cost of a target software system developed using translation will generally be higher than if the software is implemented in the target language from scratch. Some of the reasons for the increased cost include not exploiting or inappropriately using the features of the target language, and translation of the original design rather than improving the design based on experience with it. Therefore, translation will tend to result in more lines of code in the target system to be

maintained. On the other hand, maintenance costs may also be affected positively by automated translation. This is because manual translation is generally less consistent and more error prone than automated translation.

The major component of maintenance cost will be due to the level of change traffic. Constraining or eliminating non-critical changes in the target system will reduce the overall software life cycle cost.

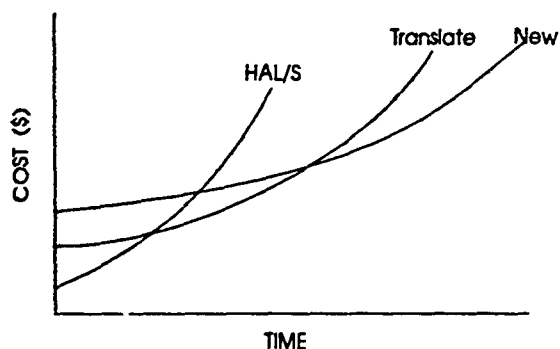
This project investigated the three different software development approaches identified in section 5.0. A thorough quantitative analysis of the maintenance costs associated with each of these development approaches was not possible. Figure 7, Expected Maintenance Cost Curves, shows that the maintenance costs of the three development approaches are expected to be different. It is for illustrative purposes only. Figure 7a shows the expected maintenance cost curve for a target system with low change traffic, while Figure 7b shows the expected maintenance cost curves for a target system with relatively high change traffic. In both cases the translated software is considered initially to be more mature than the software developed from scratch.

As can be seen in Figure 7, the amount of change traffic to the target system greatly affects when the life cycle costs of the three development approaches intersect.

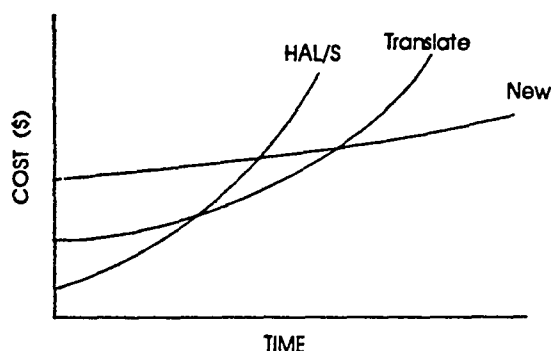
5.9 Cost Of Developing The Software Portion Of An Avionic System For A HIGHLY Shuttle Derived Vehicle

Following is an estimate of the costs associated with developing a fully operational avionic software system for an SDV. As previously discussed, it is difficult to accurately estimate the cost of developing an avionic system for an SDV without knowing the specific requirements of the system. For the purposes of this example, the following assumptions have been made:

1. The launch vehicle and avionic system are highly Shuttle derived
2. The launch vehicle is expendable
3. The launch vehicle is unmanned



a: High Change Traffic



b: Low Change Traffic

FIGURE 7. EXPECTED MAINTENANCE COST CURVES

5.9.1 Cost of Developing a Translator. To estimate the cost of developing the translator, it is first necessary to estimate its size. Table 3, presents an overview of how the size of the translator has been estimated.

TABLE 3. ESTIMATED SIZE OF TRANSLATOR

No. of trans templates identified	355
Est. percentage of total required	75%
Est. total number of templates reqd	473
No. of trans. templates implemented	203
Est. percent complete (203/473)	43%
Prototype SLOC (43% of total)	13,723
Est. total trans. SLOC (rounded)	32,000
Prototype reuse for translator	
Est. percent of reuse	40%
Total SLOC reused (40% x 13,723)	5,500
Est. SLOC to be developed	26,500

The effort necessary to develop the 26,500 SLOC required for the translator was estimated using the **CO**nstructive **CO**st **MO**del (COCOMO)². The basic model was extended to include a requirements definition phase and brief "operation" phase, during which the automated portion of the flight software translation will be performed. Maintenance of the translator will not be necessary since it will not be used after the operation phase has been completed. Finally, the 152 Hours Per Man-Month in the basic COCOMO model was adjusted to 168 Hours Per Man-Month. Table 4, Estimated Cost Of Translator, presents the results of the cost analysis.

TABLE 4. ESTIMATED COST OF TRANSLATOR

COCOMO RESULTS	
TOTAL EST HOURS	16,000
TOTAL ELAPSED TIME (MONTHS)	12
AVERAGE LEVEL OF EFFORT (EP)	9

5.9.2 SDV Avionic Flight Software Development and Maintenance. In this section the costs associated with developing a complete avionic software system for an SDV are estimated. Two alternative approaches are examined:

1. **Retain HAL/S.** This approach would modify the existing Space Shuttle Orbiter HAL/S flight software for use with an SDV.
2. **Translation.** This approach would involve a combination of automated and manual translation of all HAL/S flight software modules that would be reused, in whole or in part, for an SDV. The basic design and structure of the existing Space Shuttle Orbiter flight software would be retained.

SDV flight software sizing estimates included are based on sizing estimates generated by this project. Estimates of software life cycle costs are based on those sizing estimates and have been derived using an extended COCOMO cost model. Table 5, Allocation Of Cost By Life Cycle Phase, presents the results of this derivation and an estimated schedule for each of the life cycle phases for a "typical" real-time embedded flight software system with characteristics similar to those under analysis.

TABLE 5. ALLOCATION OF COST BY LIFE CYCLE PHASE

SW LIFE CYCLE PHASE	%COST	SCHED (MO)	
		START	DUR
REQUIREMENTS DEF	0.05	0	4
PRELIM DESIGN	0.11	4	5
DETAILED DESIGN	0.15	9	7
CODE & UNIT TEST	0.18	16	6
SOFTWARE INTEG.	0.20	22	8
SYSTEM INTEG.	0.03	30	13
MAINTENANCE	0.28	43	77
	----		--
TOTAL	1.00		120

Under normal circumstances, the effort computed using this model for a "typical" system with similar characteristics might be used to estimate the total development costs. However, for any Shuttle derived avionic system, some of the effort required to develop the software is already completed due to the ability to reuse a portion of the original requirements, design, code, test cases, etc. Table 6, Estimated Portion Already Complete Due To Reuse, presents an estimate of that portion already completed, by life cycle phase, for each of the two alternative software development approaches. The negative numbers specified for the maintenance phase indicate that these phases will require a greater than normal effort to maintain the code (i.e., both the HAL/S and "translated" Ada code will be more difficult to maintain than "Ada from scratch").

TABLE 6. ESTIMATED PORTION ALREADY COMPLETE DUE TO REUSE

SW LIFE CYCLE PHASE	TYP	RETAIN HAL/S	TRANS
Requirements Def.	0.00	0.43	0.43
Prelim Design	0.00	0.43	0.43
Detailed Design	0.00	0.43	0.41
Code & Unit Test	0.00	0.43	0.30
Software Integ.	0.00	0.50	0.10
System Integ	0.00	0.25	0.05
Maintenance	0.00	(0.50)	(0.20)

Table 7, Estimated Effort, presents the estimated effort, in Man-Months (MM), for the "typical" model and for each of the two alternative software development approaches. The amount of effort required to complete each software life cycle phase is computed as follows:

TABLE 7. ESTIMATED EFFORT (MM)

SW LIFE CYCLE PHASE	TYP MODEL	RETAIN HAL/S	TRANS
Requirements Def.	103	58	58
Prelim Design	226	129	129
Detailed Design	309	175	182
Code & Unit Test	370	210	259
Software Integ.	411	206	370
System Integ	62	46	59
Maintenance	576	864	691
	-----	-----	-----
TOTAL	2,057	1,689	1,748

(100% - % complete) * (typical model effort)

Table 8, Summary Of SW Dev & Maint Cost For Translation, summarizes the information presented for "Translation." Table 9, Constants Used, presents the constants used in this analysis.

TABLE 8.
SUMMARY OF S/W DEV. & MAINT. COSTS FOR TRANSLATION

S/W LIFE CYC. PHASE	ESTIM. EFFORT TYP MOD (MM)	% REUSE SDV	ESTIM EFFORT SDV (MM)
Requirements Def	103	0.43	58
Prelim. Design	226	0.43	129
Detailed Design	309	0.41	182
Code & Unit Test	370	0.30	259
S/W Integ.	411	0.10	370
System Integ.	62	0.05	59
	-----		-----
Total Dev. Cost	1,481		1,057
Maint. Thru Yr. 10	578	(0.20)	691
	-----		-----
10 Yr L/C Cost	2,057		1,748

TABLE 9. CONSTANTS USED

Est Total SLOC	119,000
Est STS Reuse	64,500
Est. Complete (STS Reuse/Total SLOC)	0.54
% Translatable to Ada	0.75
HAL SW Integ. Adj. for STS Reuse	0.50
HAL Sys Integ. Adj. for STS Reuse	0.25
Unit Test Portion of Code & Unit Test	0.50
Ada Maint. Advantage over HAL/S	0.20
COCOMO Result - Basic (MM)	1,302
COCOMO Adjusted - Ada (MM)	2,057
HAL Maint. During Development	6.06

The cost analysis supports the premise that automated translation may be a cost effective approach to developing an Ada avionics software system for an SDV. Although initial development costs would be lower for a HAL/S system, higher maintenance costs for HAL/S may negate these savings over the expected life of the software. It is important to note that issues related to the translation of HAL/S software into Ada were identified during this study. Some of these issues are still unresolved, and require further investigation in order to accurately assess their implications with respect to the feasibility and cost of developing shuttle derived flight software via translation.

6.0 FURTHER RESEARCH

6.1 Further Research Of Economic Issues

For translations to Ada, the primary economic area of further research is maintenance cost. Little data is available in this area. Due to the lack of data, the maintenance cost figures presented in this paper are primarily qualitative rather than quantitative.

6.2 Further Research Of Technical Issues

In regard to the HAL/S to Ada translator, the following areas need further research:

1. Tasking/Process control
2. Real-time
3. System calls
4. Input/Output

5. Use of Shuttle databases and expert knowledge to improve the translation, restructuring of the target software when compared to the source software, and placing limits on the target software data structures and types.

Information on all the above areas should be obtained before making a final decision on whether or not to translate.

7.0 CONCLUSIONS

This project investigated the issues relevant to software translation, developed a prototype HAL/S to Ada translator, translated a representative subset of Shuttle flight software, and provided a sound basis for evaluating the economic feasibility of developing a Shuttle derived avionics system using a HAL/S to Ada translator.

For the prototype translator, the following quality attributes were achieved:

1. The Ada software design is acceptable, but not optimal. The design is based on the translation templates that consider translation on a construct by construct basis, rather than a program, module, or expert knowledge basis. Automatic translation does not attempt to redesign the code. It maintains the original design and structure as much as possible. Some of the translated code will look as though it were originally implemented in Ada, and some will look like HAL/S in Ada.
2. The readability of the Ada software is good due to the use of an Ada pretty printer and deliberate design of the translation templates to support readability.
3. Although the translated code will be less efficient than code originally developed in Ada, run-time efficiency is not considered to be an issue due to use of faster processors than the current Shuttle AP-101/S.
4. Maintainability of the Ada software is fair. Maintenance effort increases due to the increased number of lines of code to be maintained.

7.1 Circumstances Under Which Translation Is Economically Feasible

Automatic translation is economically feasible under the following conditions:

1. The Source and target languages are sufficiently compatible.
2. The base of source software to be translated is large.
3. There is a high degree of similarity between the source and target systems.
4. There will be a low level of maintenance changes to the target system.
5. The expected life cycle of the target system is relatively short.

Automatic translation will probably not be economically feasible when the above conditions are not met. However, even if the systems are so dissimilar as to preclude translating the bulk of the software, it may still be economically feasible to translate part of the source software (e.g., guidance). If the target system is relatively large and similar enough to the source system, development cost of the translator and increased maintenance cost of the target system may be warranted.

7.2 Translation Cost For A Shuttle Derived Vehicle

Significant software development cost savings can result from the translation of Shuttle software for a HIGHLY Shuttle derived launch vehicle, such as Shuttle-C. The following conclusions were reached:

1. Automatic translation is feasible.
2. The Data Management System (DMS) software would be approximately 119K SLOC.
3. Approximately 46% of the DMS software would be developed from scratch.
4. A minimum of 41% of the DMS software could be translated from HAL/S.

Software development costs for using the HAL/S, translated HAL/S to Ada, and developing all the DMS

software in Ada from scratch is approximately 800, 1200, and 2800 man-months, respectively.

As shown in Figure 7 the maintenance cost for HAL/S is expected to be the highest due in part to the user (NASA) having to bear the entire cost of the HAL/S infrastructure (i.e., compilers, etc.). The maintenance cost of Ada developed from scratch is expected to be the lowest for the three alternative approaches. This is due in part to the exploitation of Ada features when developing the software, and the availability of a large base of low-cost commercial software tools for Ada. The HAL/S to Ada translation maintenance cost is somewhere between the other two approaches. For a highly derived system with a relatively short life cycle and a low level of change traffic, maintenance costs will not be a major factor. However, for systems with a long life cycle and a high level of change traffic, increased maintenance costs for the target software may far exceed any savings in development costs realized due to automated translation.

ACKNOWLEDGMENT

The authors wish to thank the following individuals for participation in the project on which this paper is based and/or for reviewing drafts of this paper: Frank Cauchon, Dan Dahlen, Don Dillehunt, Tom Healy, Jack Jansen, John Jones, Dale Keim, Kurt Kohlhase, C. Allen Lowry, John Mayer, Sandra Murray, Steve Rehagen, Gerald Rook, Deborah Terrien, Kent Tolley, Kathleen Velick. The authors apologize for any omissions to this list.

REFERENCES

1. Aho, A. and J. Ullman, Principles of Compiler Design, Addison-Wesley, 1977
2. Boehm, Barry W., Software Engineering Economics, New Jersey, Prentice Hall, Inc. 1981.
3. Ehrentfried, Daniel H., "Feasibility Assessment of Jovial to Ada Translation," Jovial Language Control Newsletter, Vol. 5, No. 1, January 1983
4. Klumpp, Allan R., An Ada Linear Algebra Package Modeled After HAL/S, JPL D-3729A, Vol. I and II, March 1989

5. Klumpp, Allan R., General Purpose Ada Packages, JPL D-6279, Vol. I and II, March 1989
6. Martin, Donald G., "Non-Ada to Ada Conversion," Ada Letters Vol. VI, No. 1, Jan/Feb 1986, pp. vi.1-72 - vi.1-81
7. MIL-STD-1815A, Reference Manual for the Ada Programming Language, United States Department of Defense AJPO), March 1983
8. Ryer, Michael J., Programming in HAL/S, Intermetrics, Inc., September 1978
9. Yellin, Daniel M., "Attribute Grammar Inversion and Source-to-source Translation," Germany: Springer-Verlag, 1988

BIOGRAPHIES

Roger S. Ritchie

Education: Was graduated in June 1985 *Cum Laude* with a B.S. in Information and Computer Science from the University Of California at Irvine (UCI). Was graduated in June 1987 with a Masters In Business Administration (MBA) from UCI. Was graduated in January 1989 with a M.S. in Computer Science from the California State University, Fullerton.

Work Experience At Rockwell International (1985 to present) Includes: Computer based simulations of the Space Station Freedom. Project lead for the Rockwell International proprietary tool: Ada Test Program (ATP). The ATP assists in the white-box testing of Ada software by determining the paths through each module of the software and generating a graphical representation of and a minimal set of test cases to test all of those paths. Project lead for the Ada Operating System (AOS) research project. This project developed a portable, cyclic based real-time executive written in Ada for embedded Ada applications. Project lead for the Shuttle Derived Launch Vehicle Software Options project. The project on which the paper is based. Manager of the Space Systems Software Analysis unit in the Software Engineering Department.

Jeffrey E. England

Mr. England is Manager of Space Systems for the Systems and Software Engineering Division of the Aerospace Systems Group of Intermetrics, Inc. located in Huntington Beach, California. He received a BA degree in Quantitative Methods from California State University at Fullerton in 1974. Mr. England is a principal architect of the Build, Integration, and Mission Reconfiguration System for the Space Shuttle Backup Flight System and has substantial experience in the development of testing and analysis tools for the Ada language.

ADA TRANSITION RESEARCH PROJECT (A SOFTWARE RE-ENGINEERING EFFORT)

Glenn E. Racine (AIRMICS) Reginald Hobbs (AIRMICS) Richard Wassmuth (SDC-A)

SUMMARY

This paper describes a re-engineering effort recently conducted by the Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS) and Software Development Center - Atlanta (SDC-A), both components of the Army's Information Systems Engineering Command (ISEC). The project re-engineered a Management Information System (MIS) written in COBOL to a new system written in Ada. The paper covers the following: background, project objectives, approach, brief description of the system that was re-engineered, reverse engineering, redesign strategy, design process, implementation aspects, results, issues and current efforts, and conclusion.

BACKGROUND

The Army's Information Systems Command (USAISC) maintains over one hundred STAMIS (Standard Army Management Information System) applications in the major functional areas of logistics, personnel, finance, and communications. Most of these systems are written in COBOL, are batch oriented, and some have been operational for more than twenty years. With these systems, it is difficult to embrace and incorporate new technology. New technology, in the form of Ada, has been mandated (by DOD) for all new system development. Additionally, current software development for new systems is lengthy and maintenance is expensive. In fact, some estimates state maintenance consumes sixty to eighty per cent of life cycle costs. Because of the large investment in the systems being maintained, it is desirable to leverage current state-of-

the-art technology, tools, and modern software engineering principles into these old applications.

OBJECTIVES

A number of objectives were key to this project. We wanted to assess the use of CASE (Computer-Aided Software Engineering) tools in the reverse engineering, design, and Ada implementation phases. Additionally, we wanted to compare Object-Oriented Design (OOD) versus structured design (also known as functional decomposition). Since the system was to be written in Ada, one of the objectives was to determine what the issues were in implementing Ada. Training would be a key factor because we were using CASE, OOD, and Ada for the first time. At the conclusion of the effort, we also wanted to compare the old system with the new, re-engineered system.

APPROACH

The first step was to select a STAMIS to re-engineer. The STAMIS had to be small, capable of being modified, and representative of most STAMIS. The next step was to reverse engineer the system to document the functionality of the system. CASE tools were used to help document the requirements, but this was largely a manual task. The functional description was then reviewed with the Functional Area Specialist and some corrections were made. The team was divided into two groups, one to do functional decomposition and the other to do the OOD. The design teams used the same function description to define the functional requirements of the system. The designs were compared to determine the design for implementation. The programmers were given eight weeks of Ada training and the design that was selected for implementation.

SYSTEM DESCRIPTION

The Installation Materiel Condition Status Reporting System (IMCSRS) was selected. IMCSRS was an application of manageable size with all the characteristics of most of the STAMIS systems. It was a 20 year old, batch oriented system, containing 10,000 lines of COBOL code. The system ran at 40 installations on a mainframe computer. The size of the system was limited so that the concentration would be on the project objectives, rather than simply coding a large system.

IMCSRS consolidates equipment status information at each installation and extracts data needed to help managers evaluate the status, readiness posture, and condition of selected items of equipment. It automates the front side of a Department of Army Form (DA Form 2406 - Materiel Condition Status Report) and forwards the information, via AUTODIN (Automated Digital Network), to the Materiel Readiness Support Activity (MRSA). MRSA loads the information into the Readiness Integrated Database. Consolidated information is sent via AUTODIN to FORSCOM and TRADOC - two major Army commands. Printed reports on readiness posture and maintenance status of installation equipment are produced at the installation/division level.

REVERSE ENGINEERING

[The reverse engineering process was conducted by a team from Georgia Tech. Detailed results of this process are discussed in ¹. Information in this section has been extracted and summarized from that report.]

After selecting the STAMIS, the next step was to reverse engineer the system. "Reverse Engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction."²

Reverse engineering is a critical event in the re-engineering process. As systems are maintained over a number of years, the quality of the program struc-

ture tends to degenerate and the supporting documentation no longer reflects the operational system. Reverse engineering is largely a question of recognizing design decisions in software and constructing a representation of these decisions from which a redesign can be performed. Constructing a high level representation (functional description) of existing software facilitates the construction of a new and more appropriate design. Our experience on this project points to reverse engineering as having two major levels of abstraction, a snapshot view and a concept view. The snapshot view attempts to capture the requirements and functionality inherent in the current system. The snapshot results are then reviewed with a functional area specialist to come up with a concept view of the system. These two-views are discussed next.

Snapshot View. Reverse engineering involves creating a higher level representation of source code. It is appropriate to select graphical design representations to serve as media for expressing the results of the reverse engineering process. Four representations were used in this effort and they differ in the level of abstraction described and the facet of the source code represented. They are:

1. System Level - Context diagrams and English text.
2. Cycle/Program Level - Data Flow Diagrams (DFD).
3. Program Level/File Structure - Jackson Diagrams.
4. Program Level/Algorithmic Structure - The structure of the code is organized in terms of design decisions. (Design decisions convey the use of various programming language features and coding techniques. This facet attempts to divide the design decisions into classes based on the type of abstraction they provide. The classes that are useful include composition/decomposition, encapsulation/interleaving, generalization/specialization, representation, data/procedure, and modality. A discussion of these classes is in ³.)

These four steps also constitute the four major steps of the reverse engineering method. Step one data

should be derived primarily from information external to the actual system code – system documentation and, possibly, interviews. Step two data should be derived from system documentation and verified by code examination. CASE tools exist that can be used to construct the DFDs. The purpose of the Jackson Diagrams in step three is to describe the internal file structure. Some CASE tools have editors to construct Jackson Diagrams. In some cases (documentation not clear, code clarity is inadequate) a more detailed level of the program is necessary. In these cases a line by line analysis may have to be performed, as in step four. A technique to perform this analysis is called Synchronized Refinement. It is driven by the detection of design decisions found in the code. The end result of Synchronized Refinement is an annotated structural description of how the program function is accomplished in the code. This process was used in our effort and was mainly a manual process. Tool support, at present, is limited. Our experience on this project points for the need to have a functional specialist validate and verify the output of this process.

Concept View. This view is the highest level view of the system. It is what the new functional description is based on. One of the issues that arises is how significant are the algorithms, cycles, and programming decisions which were done on the system to be redesigned. The concept view should target on what the new system should do, and in what environment. The medium for this is a System/User Chart and is a high level graphical representation of what the new (re-engineered) system should do. It is then refined to lower levels of detail. Based on this effort, it appears this process is both top down and bottom up. Top down in the sense that the concept view can be developed through interaction with a functional area specialist while the bottom up view can be accomplished by creating the snapshot view. The difficulty lies in combining the two views. One strategy could be to require the output of the snapshot view to be in a prescribed format, e.g., DOD-STD-7935A. This is one area we plan to address in the future.

REDESIGN STRATEGY

Design Issues. One of the fundamental issues raised in the redesign effort is the question of program structure. COBOL lends itself to a functional architecture. Major system activities are assigned to cycles and decomposed into programs; even program functions are broken down into paragraphs. In all cases, the unit of decomposition is an activity or process.

The Ada approach is quite different. Ada allows for the decomposition of programs into packages, where each package corresponds to a major system object. Ada, however, is not an O-O language. What we did was take O-O design and looked to the language for constructs to support the O-O Design concepts. A system can very easily be written functionally in Ada. Translators also may exist to convert COBOL to Ada. However, design concepts are difficult to capture with translators and changing systems to reflect O-O implementations is quite different.

With this in mind, there are some guidelines that can be suggested to facilitate the process.

1. Begin with the textual description produced by the reverse engineering process. Object-Oriented Design begins with the composition of a textual statement of system requirements, so it is natural to begin with the one already written.
2. Use the input and output files and reports as initial candidates for objects. Look for typical situations normally handled in COBOL by functional means such as READ/TRANSACTION/WRITE. Invert these by devising an object to hide the file structure. Add operations to initialize the object (READ), to update its state (TRANSACTION), and to save its value (WRITE). The transaction file itself may be a candidate for an object.

NOTE: One risk of looking at old files and records is carrying old design decisions into the new design. An alternative, or perhaps parallel activity, is to develop a chart (i.e., System/User Chart) that contains the real world objects that interface to the system.

These objects are then compared to the functional description.

3. The approach described in the preceding paragraph may occasionally run into resource limitations; i.e., it may not be possible to contain an entire file in memory at one time. In this case, the Jackson Diagram can be used to isolate repetitive batches in files (repeating groups of records). A new subsidiary object can be constructed to hide the batch definition. If batches can be processed sequentially, the file object can periodically request that the batch object "renew" itself (obtain a new batch).

4. The redesigner should also be on the lookout for "second order factoring." This occurs when the initial system decomposition leads to objects that share functionality and structure. For example, if objects exist to hide the details of the various output reports, many of them may need to perform their own pagination. This suggests that a new "page" object be defined, responsible for taking in and counting lines and periodically ejecting pages. The page object can, in turn, be used by the report objects as a resource.

5. Another observation relates to the overall execution structure of the STAMIS. The mainframe environment and the use of JCL lead themselves to structuring the system in terms of separate job steps and lots of intermediate files. These are artifacts of the environment and not system requirements (unless there are some non-functional operational constraints that are not obvious from the documentation). We suggest that the redesign proceed without these limitations.

6. A related concern is the user interface. Although there is a desire, for comparison purposes, to construct a system with identical functionality, there are some improvements that could be made concerning how the system is controlled. For example, the selection of the reports to be generated naturally suggests a menu interface. A direct manipulation user interface should be considered as an alternative to the JCL approach.

Implementation Considerations. The Ada language provides a variety of features that go well beyond the power of COBOL. Among those that should be considered for use are the following:

1. Subprograms. The IMCSRS STAMIS made no use of subprograms. PERFORM clauses were used when decomposition was required, but the Ada PROCEDURE and FUNCTION constructs should be taken advantage of.

2. Packages. The primary structuring unit of Ada is the PACKAGE. PACKAGES naturally correspond to objects in Object-Oriented Design.

3. Exceptions. The Ada EXCEPTION mechanism provides a way to remove the handling of unusual or unexpected situations from the main stream of the source code without compromising thorough error checking. COBOL constructs such as the ON END clause of the READ statement, the INVALID KEY clause of the WRITE statement, and the production of SYSCOUT error messages should be considered as candidates for handling by the Ada EXCEPTION mechanism.

4. Tasks. Ada provides a comprehensive tasking mechanism. It has come under some criticism for its difficulty in handling the response time needs of embedded systems. However, it does provide a portable mechanism for dealing with a certain class of problems. The question is, are those kinds of problems likely to arise in STAMIS's. Probably not. However, there are some situations where parallelism is possible, and these may serve as candidates for tasking. For example, several reports could be produced simultaneously. If several files need to be updated independently before being merged (as was the case with several of the files in IMCSRS), these could be handled in separate tasks. For example, if a file is handled by independently processing batches of records, each batch might be handled as a separate task.

5. Generics. Another interesting feature of Ada is the GENERIC. This allows a set of subprograms or PACKAGES to be described by a parameterized template. Multiple instances can be created possi-

bly differing by the data used, or the function handled. For example, a generic QUEUE package could be defined and parameterized by the type of element being enqueued. QUEUE's of INTEGERS and QUEUES of job requests could then be "instantiated."

Generics can also serve as a method for reuse. That is, a generic PACKAGE, such as the QUEUE package described above, can be used in many different systems. It is suggested that as the redesign/redevelopment process emerges, that an explicit effort be made to recognize, characterize, develop, and employ reusable components.

There is one final suggestion to be made concerning the use of Ada. Ada has been criticized as being overly complex and, hence, unreliable.⁴ Ada certainly is a large language with many powerful and interesting features. One way of dealing with the complexity is through the use of strict coding standards that limit the features used and the modes of expression. For example, many shops prevent the use of the USE clause. Conventions should be agreed on before development begins and used rigorously. One counter to the complexity argument is that implementation is easy if you have a good design. If the system is designed properly, the flexibility a programmer needs is available, however, overall program structure cannot be altered if the specifications are protected.

DESIGN PROCESS

One of the major issues explored during this project was which method—object-oriented or functional decomposition—is the more appropriate for this redesign. Redesigning the system using any structured method would produce a positive improvement in the systems' functionality due to the maintenance problems cited earlier. OOD is a maturing technology and lacks a wealth of empirical data that supports its use for STAMIS applications. Functional decomposition is the more traditional and accepted design method. To explore this issue and gain insight into the software development process, project members were divided into two teams. One team re-

designed IMCSRS using the object-oriented approach and the other using Functional Decomposition. Also of concern during the project, was gauging the impact of automated tools on the design process. There was a need for defining criteria to select appropriate tools that would assist the developer while minimizing the learning curve necessary to make them effective. Assessing some of the essential requirements for Ada training to prepare programmers and developers for transitioning systems was another objective of the project. Following is a brief discussion of CASE tools, functional decomposition, and object-oriented design.

CASE. CASE (Computer Aided Software Engineering) includes automated tools used within the software development life cycle. In the beginning of the technology, CASE tools were primarily used to assist in the generation of graphical representations of a system. The underlying semantics and rules for a particular method were being done for the most part by the developer. Current tools incorporate features such as project management, requirements tracking, or code generation and employ software engineering methodologies such as structured analysis, structured design, and OOD. Although CASE tools are still a maturing technology (the industry has been around for 15 years), there are many CASE products on the market. The key in selection is to determine at the beginning of the project what characteristics are critical to the tasks to be accomplished. Once this is determined, the various tools can be compared against these critical success factors until the tool meeting most, if not all, factors is selected.

The training of the developers who will use a particular CASE tool in a design project of this type is a major consideration in choosing the tool. While choosing a tool that applies and validates a particular method will help ensure standardization and the use of good software engineering techniques, there are drawbacks. A vendor-sponsored training program will frequently focus on the mechanics of the CASE tool instead of teaching the underlying method. Programmers who learn structured analysis in

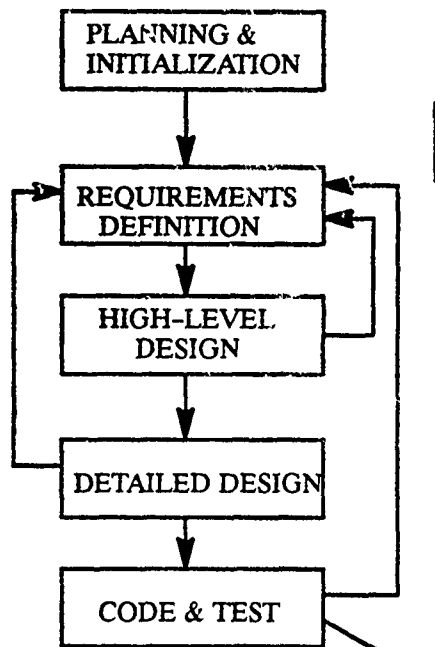
the context of a particular CASE tool may have a tendency to think of analysis as an operation to be performed with that tool, rather than as an intellectual activity that the tool supports. It is very important that the developers go through some formal instruction in using software engineering techniques.

The "waterfall model" of the software development life cycle views system development as a series of steps, flowing downhill from requirements definition, through analysis, design, coding, testing, implementation, and maintenance. The problem with the waterfall model is that it doesn't accurately describe the way software is developed in the real world. System construction is normally an iterative effort. In the following diagram (Figure 1), is an interpretation of the impact of CASE tools on the traditional software development life cycle. One of the reasons that CASE tools can have a significant impact on programmer productivity and the creation of well designed systems is the elimination of the iteration that often occurs in completing the requirements definition. By constructing a graphical representation of the system at a high-level of abstraction, the system can be modeled in such a way that the user requirements are adequately depicted. Refining this model can then be based on user input (along with information from the functional requirements document) to create a "real world" representation of the problem being solved. The use of prototyping tools at this level further validates the accuracy of the CASE model. For this transition project, no prototyping tools were used. The output from the high-level design can then be used to create a detailed design of the system. Different CASE tools are used to implement each stage of the life cycle. In order for tools to be fully integrated, two actions need to be completed—the creation of a common repository and the definition of translation mechanisms for transitioning design representations through the different stages in the life cycle. The development of Integrated CASE tools that support a repository and this type of translation between representations will further automate the life cycle process.

Functional Decomposition. Redesigning IMCSRS using functional decomposition techniques was one of the stated goals for this transition effort. Functional decomposition, as the name implies, analyzes and designs systems from a functional perspective. The procedural characteristics are the primary focus; data is considered during the later stages of development. Functional decomposition can be divided into two distinct phases: Structured Analysis and Structured Systems Design. Structured Analysis involves determining a structural model of an application and its corresponding structural specifications to outline the problem space ("What" we are attempting to do). Structured Design techniques map the problem space to a solution space, creating a new system containing the same components and process relationships outlined during the analysis of the system ("How" we are to do it). The structured analysis of a system precedes its design; the structural specifications are direct input for the design phase. The development of graphical structural specifications produces a model of the system that is very concise and easy to interpret. The main building block of structured analysis is the Data Flow Diagram (DFD). A DFD depicts the active system processes and their interrelationships. A process is an event within the system where some transformation of data takes place. The diagram is organized so that data can be followed from process to process, which establishes how the processes communicate. A DFD shows the source, destination, and location of the data. Unlike conventional flowcharts, DFDs don't show flow of control or procedure sequences. The temporal sequence of events within a system (i.e. which process happens first, second, etc.) are not described on a DFD. One of the most difficult tasks during the structured analysis phase is reducing system activities to processes that transform data.

In the structured design portion of a system development, the structured specification generated in the analysis phase is used as a road map towards the new design. By using the structured specification as

TRADITIONAL DEVELOPMENT



RE-ENGINEERING DEVELOPMENT USING CASE TOOLS

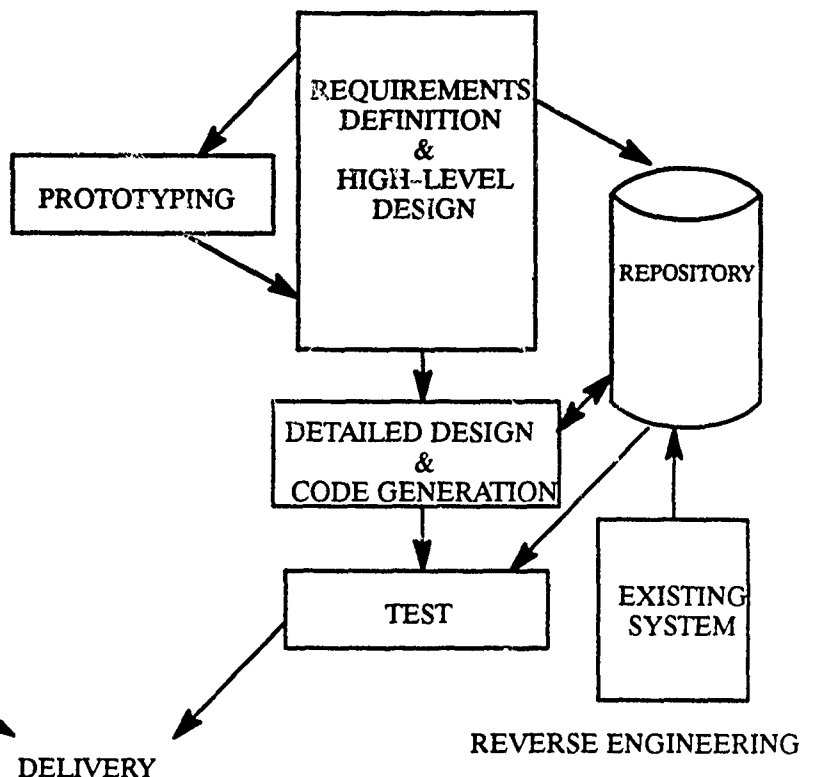


Figure 1
Impact of CASE Tools in the Software Development Process

a well-defined statement of the problem, structured design allows the form of the problem to guide the solution. The design itself is not the final system but a plan for implementing the new system. Structured design can also help outline a set of criteria for evaluating the proficiency of a design solution as it relates to the original problem. Structured design derives a simplified, graphical representation of a system by partitioning the functionality and organizing hierarchical relationships. Partitioning systems modularizes function; effectively isolating functions from updates and changes in different portions of a system. The DFDs of the structured analysis phase are converted into structure charts in the design phase through transform analysis. Transform analysis is a strategy for deriving a first-cut structure chart from the analysis of a system. At present there is no automated, heuristic process of moving from DFDs to structure charts. The process is not

necessarily intuitive and requires many iterative steps of refinement. The beginning of the transform analysis process involves determining the portion of a DFD that contains the essential functions of a system. This central transform will outline the basic functions that must be accomplished within a system and lead directly to the development of a structure chart. The lack of a seamless integration strategy between structured analysis and structured design is sometimes viewed as a major limitation of using the method. Some software developers say that analysis and design should always remain separate; otherwise design considerations begin influencing analysis decisions before it is appropriate to consider them.

Object-Oriented Design (OOD). OOD is a new approach to system modeling that evolved as a natural consequence to object-oriented programming. In

the object-oriented environment, data are primary, procedures are secondary, functions are associated with related data, and a bottom-up approach is used for development. This method is diametrically opposed to the way software development has been performed in the past, where system characteristics were defined from a functional perspective. Terms such as information hiding, "black box" design, or encapsulation are also used to describe OOD because information concerning detailed design or implementation need not be known by multiple developers constructing a system designed in this manner - only the interface specification need be known. Perhaps the strongest features of OOD include increased reusability, maintainability, understandability, and partitioning of the problem so that several programmers can implement the system concurrently and with little communication if the interfaces have been well defined. An object is defined as an entity (file, device, organizational unit, events, physical locations, or any abstract object) that is itself defined by a set of common attributes and services or operations associated with it (sort, retrieve, read, write, etc). Identifying the objects is the most difficult step in the design method. Abstract objects, such as speed and time, are particularly hard to identify since they do not physically occur within the system. Each object has associated with it attributes that define the object. The attributes are not graphically represented but are implemented as fields in the defining data structure. Objects have operations, both active and passive, associated with them. Active operations alter an object's state and the value of its attributes. Passive operations do not alter an object but usually return the attribute values to the calling procedure. The operations are created either as visible operations which are accessible by other objects or local operations that can only be used within the object. The object-oriented design process involves the following phases:

1. Identify the objects and their attributes.
2. Identify the operations acted upon and required of each object.
3. Establish the visibility of each object in relation

to other objects.

4. Establish the interface for each object.
5. Implement each object.

The objective of the design phase of the project was to produce a detailed design of the system which was ready for implementation in Ada. Using the newly developed functional description as a baseline, the system was redesigned by two separate teams; one developed a Functional Design, the other an Object Oriented Design.

The two designs were compared and the Object Oriented Design was selected as the design for implementation. The Object Oriented Design was selected because it was simpler to understand, appeared easier to implement and promised reduced maintenance effort during the life-cycle. Object Oriented Design models the real world. The basis for the design method are real world objects. These objects are very stable creating a solid foundation for the system. In this system, the objects were the Installation, Units, Reportable Equipment, the DA Form 2406 and Reports. These objects are very modular and easily separated for implementation and maintenance.

Ada is perfectly able to support large functionally designed systems. It is not the language that has a problem with functional design. In many cases, it has been structured design that was difficult to work with on large systems. OOD was developed so no one programmer must see the entire system to design it. You see only what one object sees at a time. The language was applied to the technique. Again, Ada was not developed with OOD in mind.

IMPLEMENTATION

The Object-Oriented Design was so easily understood that it was used as a basis for final validation of the functional requirements by the proponent agency (PA). After viewing the new system design, the PA submitted an Engineering Change Proposal (ECP) to implement the new design as a production system, to replace the old mainframe version. Upon receipt of the PA's request to move IMCSRS from the mainframe to a desktop computer, it became a

production STAMIS to be extended world-wide. New functionality was integrated into the functional description and current design.

The CASE tool automatically generated Ada specifications from the design. These specifications served as the framework in which the programmers would implement the system.

Two programmers received 8 weeks of Ada training. Four weeks of basic Ada training by Software Development Center - Lee and four weeks of advanced training at Keesler, Air Force Base.

Components from the Reusable Ada Packages for Information Systems Development (RAPID) library were received from Software Development Center - Washington. Software Development Center - Atlanta developed its own reusable components prior to coding the system.

The system was coded and tested by two newly trained Ada programmers at Software Development Center - Atlanta in five months. Total work effort was seven man-months.

An errorless software acceptance test (SAT) was conducted at Fort Stewart, GA. Several weeks later, the system was extended by mail to 40 installations.

RESULTS

The resulting system is now running on existing PC's (personal computers) instead of a mainframe at a regional data center. It is written in Ada, is interactive, and turnaround has drastically improved.

These benefits are primarily the result of the re-engineering. The change of business practice was the real benefit. Processing costs have been reduced through downsizing. Reliability has been improved, and we expect that maintenance costs will be dramatically reduced. But more importantly, we have improved the efficiency of users at each of forty sites.

Comments from three of the forty sites where the new system is running point to the efficiency of the new system. Their comments were:

"What has taken us up to 2 1/2 weeks to do with the old system, we completed in 35 minutes." PANAMA

"Our average time from start to finish was 1 1/2 weeks, on a monthly cycle. We completed it in 4 hours. It took 2 1/2 weeks on a quarterly cycle. The new system took 2 days. The new software drastically reduces effort." FORT McCLELLAN

"We made 5 trips between 4-5 miles to the DOL and DOIM and used 2-3 operators on the old system. With the new system, we use one person for input and make one trip to the DOIM." FORT CAMPBELL

ISSUES & CURRENT EFFORTS

One of the questions that arises is whether the success of this effort is repeatable. The MMCSRS (MACOM Materiel Condition Status and Reporting System) is currently being re-engineered and the results will be compared against the IMCSRS effort.

AIRMICS is associated with a National Science Foundation Research Center (Software Engineering Research Center co-located at Purdue University and the University of Florida) that has a project⁵ to develop a metrics approach for analyzing software designs which helps designers engineer quality into the design product. The design metrics project has developed a design quality metric that analyzes the external and internal design components. We are applying these design metrics against the old and new systems to serve as a basis of comparison.

Finally, we are performing a cost/benefit analysis study to determine the effectiveness of our approach and also to use it as the basis for determining which applications should be selected for future re-engineering activities.

CONCLUSION

CASE tools are very helpful to the design process. Training is extremely important. Training in CASE is probably no different than learning any new software tool. Training in Ada requires not only training in the syntax of the language but also requires an understanding of software engineering principles as a prerequisite. Understanding object-oriented concepts requires somewhat of a mind set change and some experience to be able to maximize its use. O-O design is an effective and efficient way to design systems and appears to be workable for STAMIS-type applications. We also expect significant efficiencies to be gained from the re-engineering process—specifically in maintenance and operations—because they automate what, in the past, has been, in most instances, a manual process. CASE allows for rapid change in design (easy to try new ideas), promotes software engineering principles, enforces Ada construction rules, and aids in standardizing the development process.

REFERENCES

1. Rugaber, S., and Kamper, K., "Design Decision Analysis Research Project Final Report," Software Engineering Research Center and School of Information & Computer Science, Georgia Institute of Technology, January 28, 1990.
2. Chikofsky, E.J., and Cross II, J.H., "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software vol. 7, no. 1, January 1990.
3. Rugaber, S., Ornburn, S., and Leblanc, Jr., R., "Reengineering Design Decisions in Programs," IEEE Software, vol. 7, no. 1, January 1990.
4. Hoare, C.A.R., "The Emperor's Old Clothes," Communications of the ACM, vol. 24, no. 21, pp. 75-83, February 1981.
5. Zage, W., and Zage, D., "Relating Design Metrics to Software Quality: Some Empirical Results," SERC-TR-71-P, May 1990.

ACKNOWLEDGEMENT

The authors would like to thank Dan Hocking for his help in editing the paper.

AUTHORS

Glenn E. Racine, AIRMICS(Army Institute for Research in Management Information, Communications and Computer Sciences), 115 O'Keefe Building, Georgia Institute of Technology, Atlanta, GA 30332-0800. Mr. Racine received a BA in mathematics from Western Kentucky U. in 1969, an MS in Business from Virginia Commonwealth U. in 1978, and an MS in Information and Computer Science from Georgia Tech in 1985. He is currently the Chief of the Computer and Information Systems Division at AIRMICS and his current areas of responsibility are applied research activities in Software Engineering and Very Large Data Bases.

CPT Richard Wassmuth, SDC-A (Software Development Center - Atlanta), Fort Gillem, GA 30050-5000. Cpt Wassmuth graduated from West Point in 1982 with a BS in Engineering. He received an MS in Computer & Information Sciences from the University of Florida in 1988. He has worked in the Strategic Communications Area in Germany and is currently working as a Software Engineer/Teleprocessing Operations Officer at SDC-A.

Reginald Hobbs is a Computer Scientist in the Computer and Information Systems Division at AIRMICS. He has experience in Information Systems from HQ Air Force Space Command, Cheyenne Mountain Complex, Colorado and as a Systems Administrator/Systems Programmer at the Army Research Institute, Alexandria, Virginia. Mr. Hobbs has a B.S. in Electronics from Chapman College. He has research interests in the areas of software development methods, Computer-Aided Software Engineering (CASE) tools, database management systems, and software metrics.

DESIGN METRICS THROUGH A DIANA BASED TOOL

Wayne M. Zage and Dolores M. Zage
Computer Science Department
Ball State University, Muncie, IN 47306

Dale J. Gaumer and Michael J. Meier
Magnavox Electronic Systems Company
Fort Wayne, IN 46808

Abstract -- This paper discusses a metrics approach for analyzing software designs which helps designers engineer quality into the design product. These metrics gauge project quality as well as design complexity at all times during the design phase. The metrics are developed from primitive design metrics which are predictive, objective and automatable. The architectural design metrics used are comprised of terms related to the amount of data flowing through the module and the number of paths through the module. A detailed design metrics component takes into account the structure and complexity of a module. To automate the calculation of the design metrics in the Rational environment, DIANA (Descriptive Intermediate Attributed Notation for Ada) was utilized. Provided in the environment are packages allowing for the traversal and retrieval of the DIANA structure. By combining the defined packages with customized packages, an Ada design metrics analysis tool was developed. This paper will discuss our design metrics and their automation at Magnavox. Empirical results will illustrate the metrics' success in identifying stress points in a software design and demonstrate their relationship to the quality of the resulting software.

Index Terms -- design metrics, development process, quality assessment, DIANA

Introduction

Software developers should be able to infer more about the software they are developing during the design process. By computing metrics at various times during the development of design, project personnel can

identify favorable and unfavorable design trends. This information could help managers and software developers determine the better design when alternative choices exist, as well as identify stress points which may lead to difficulty during coding and maintenance.

It has been recognized that most of the important structural decisions have been made irreversibly by the end of architectural design.² Researchers have sought a metric which would identify problematic components early in the life cycle. Studies have shown that approximately 20 percent of a software system is responsible for 80 percent of the errors.¹ It is possible that such error-prone modules exhibit some measurable attribute to identify them as design stress points. Locating these stress points early will reduce the cost of development and ultimately lead to more reliable software.

The goal of this design metrics research is to improve the software development process by providing timely information about the design of a developing system in order to guide the designer to a better software product. Our objective is to develop for a design G a design quality metric $D(G)$ comprised of an *internal* and an *external* design quality component. This composite metric will be used to predict potential quality and complexity of the system being developed.

Our Design Metrics and Related Findings

The Design Metrics Research Team at Ball State University has developed a design metric $D(G)$ of the form

$$D(G) = k_1 D_e + k_2 D_i$$

where k_1 and k_2 are constants. In this equation, the external metric D_e is defined as

$$D_e = \text{weighted inflows} * \text{weighted outflows} \\ + \text{number of possible paths}$$

and the internal metric D_i is defined as

$$D_i = w_1(CC) + w_2(DSM) + w_3(I/O)$$

where

CC (Central Calls) are procedure or function invocations

DSM (Data Structure Manipulations) are references to complex data types

I/O (Input/Output) are external device accesses

and w_1 , w_2 and w_3 are weighting factors.

The metrics D_e and D_i are designed to offer useful information during two different stages of software design. The calculation of D_e is based on information available during architectural design, whereas D_i is calculated after detailed design is completed. (See Figure 1.) In architectural design, information such as hierarchical module diagrams, data flows, functional descriptions of modules and interface descriptions are available. After completing detailed design, all of the previous information plus the chosen algorithms, and in many cases either pseudocode or a PDL representation for each module, are available.

After architectural design is completed, a software designer would calculate D_e values for the modules (or named entities) in the system. Then, based on the outcome of these calculations, the designer would either proceed to detailed design or revise the architectural design of the software, and then recalculate the D_e values. Once this iterative process is satisfactorily completed, detailed design begins, after which D_i values are computed. Based on the outcome of these calculations, one either proceeds to coding or revises the detailed design, or in extreme cases, goes back for further revision of the architectural design. The point of calculating D_e and D_i at these separate stages of design is to use the information currently available about the developing system to provide an indication of project

status and complexity "early" in the life cycle. In the next several subsections, we will show the degree to which D_e , D_i , and $D(G)$ accomplish this task.

The Design Metrics Test Bed

The design metrics test bed contains a number of university-based projects written for industrial clients as well as software developed professionally. This paper will focus primarily on the metrics results for the professionally developed software. In 1986, the Standard Finance System Redesign (SRD-II) contract was awarded to the Computer Sciences Corporation (CSC) at Fort Benjamin Harrison in Indiana. SRD-II is part of the Standard Army Financial System Redesign (STANFINS-R, which we shall refer to simply as STANFINS) software system. It contains approximately 532 programs, 3,000 packages and 2,500,000 lines of Ada. Our design metrics study was completed on 21 programs (approximately 24,000 lines of code) which were selected by CSC development teams.

The design of STANFINS began approximately six years before this study. To verify that the design was an exact representation of the Ada source code, the design was extracted from the existing code instead of from the design specifications. Tools were developed to collect the elements of the design needed to calculate our design metrics. Information on error occurrence was also collected. Another program was written to execute a CMVC (Change Management Version Control) history search on target packages to map the changes to the name of the modules in the abstracted design. These changes occurred as part of the second phase of the software qualification testing cycle.

Table 1 includes a number of characteristics of the STANFINS data. The first column displays the number of generations in the development of the 21 programs. (False generations, such as a check-in after no changes were made, were ignored.) Also listed in the table are the total source lines of code, the number of errors found, and the errors per KLOC. The number of modules and the number of modules with errors are also displayed.

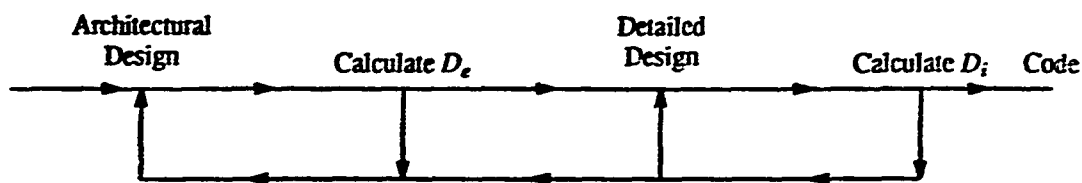


Figure 1: The Calculation of D_e and D_i in the Life Cycle

Table 1: STANFINS Data and Error Report

Generations	LOC	Errors	Errors/ KLOC	Modules	Modules w/Errors	LOC Avg	D_e Mean	D_e Std Dev
162	23732	3002	126	2384	314	34	202	372

Empirical Results

The goal of the design metrics calculations is to identify error-prone modules during the design phase of software development. The method employed is to calculate the particular design metric for each module under consideration. Then outliers (or stress points) are identified as those modules whose metric value is more than one standard deviation above the mean for that metric over all of the modules considered. Later, when error reports are available, we determine to what extent the stress-point modules are identified as error-prone.

We had earlier obtained excellent results on finding error-prone modules in relatively small university based projects (2,000 to 30,000 lines of code) by simply calculating the D_e values. The 12% of the modules identified as stress points by D_e contained 53% of the known errors over all the systems in our database.³ We then needed to determine if these results would hold as we tested our metrics on large-scale, industrial-based software.

D_e was also found to be an effective predictor of error-prone modules in the STANFINS software. As shown in Table 2, for all modules considered, D_e identified 37% of the errors while only highlighting 5% of the modules. (Recall that D_e uses only informa-

tion available during architectural design.) Overall, D_e correctly identified modules as error-prone 60% of the time.⁴

Table 2: D_e Performance on STANFINS Data

Modules Highlighted	5%
Highlighted Modules With Errors	60%
Errors Found	37%
Error Modules Not Found	76%
Errorless Modules Highlighted	40%

We also compared the error versus errorless modules with respect to their D_e values in this study. The results are contained in Table 3. Note that the D_e mean for error-prone modules was more than six times the mean for errorless modules!

Table 3: Error vs. Errorless Modules

	Error Modules	Errorless Modules
Number of Mods	314	2070
D_e Mean	355	54
D_e Std Dev	666	375

During our analysis, we have found that extreme outlier modules (with respect to D_e and D_i) exist in large-scale software. For

example, on the STANFINS data we found the number of modules below the D_e mean was more than five times the number of modules above that mean (1991 vs. 393). Therefore, those modules above the mean had relatively extreme D_e values. These modules, with their extreme metric values, adversely affect our stress point cut-off values. This led to our *X-less design metric algorithm* in which we remove X such modules from the calculation of cut-off values to dramatically improve the results. For example, we found that if enough extreme outliers are taken into account when calculating D_e so that 33% of the modules are identified as stress points, then 97% (2912) of the errors in the STANFINS system were found.⁴ The trade-off is that the false positives have grown to 65% of the highlighted modules at this point. However, practitioners have remarked that this is a small price to pay for the use of the simple design metric D_e , early in design, to find such a high percentage of errors while still only reviewing one-third of the modules.

Using the *X-less* algorithm, we once again compared D_e 's performance on the STANFINS data to its performance on the university database. As stated earlier, in the university study, D_e targeted 53% of the errors while highlighting 12% of the modules. On the STANFINS data, D_e performed even better than on the university database by targeting 67% of the known errors (2011) when identifying 12% of the modules (using the *X-less* algorithm) as stress points.⁴

D_i also performed well when applied to the STANFINS project. (See Table 4.) In that study, D_i alone identified 54% of the errors while only highlighting 10% of the modules. Note also that 54% of the errors were contained in 31% of the modules containing errors, and thus high concentrations of errors were present in these modules. The false positive rate for these data was 22%, meaning that for every five modules highlighted, approximately four contained errors. We also found that if 20% of the modules are identified as stress points (using the *X-less* algorithm), then 74% of the errors were found. Moreover, for these data, we found that 27% of the variance in errors was due to the DSM count.⁴

Table 4: D_i Performance on STANFINS Data

Modules Highlighted	10%
Highlighted Modules With Errors	78%
Errors Found	54%
Error Modules Not Found	69%
Errorless Modules Highlighted	22%

The $D(G)$ metric was very successful in detecting the STANFINS modules with errors, with false detection of very few error-free modules. On this test bed, 74% of the errors were detected for modules with 10 or less errors, 82% of the errors were detected for modules with 11 to 20 errors, and 100% of the errors were detected for modules with 21 or more errors.⁴ In other words, modules with high concentrations of errors were always identified as stress-points by $D(G)$. These results are summarized in the graph shown in Figure 2.

Tool and Software Analyzer Development

Tools must be available to successfully integrate the use of software metrics into the software development process. Tools insure consistent measurements and minimize the interference with the existing work load.

Many large projects which resided or could be ported to the Rational environment were offered as data for this research. To make this analysis more efficient and complete, a Design Metric Analyzer was created. To automate the calculation of design metrics in the Rational environment, DIANA (Descriptive Intermediate Attributed Notation for Ada) was utilized. DIANA is the standard intermediate representation of Ada programs. DIANA was developed to provide a common interface between different phases of the Ada compiler. The DIANA structure stores the syntactic and semantic aspects of Ada designs and programs in the Rational environment.

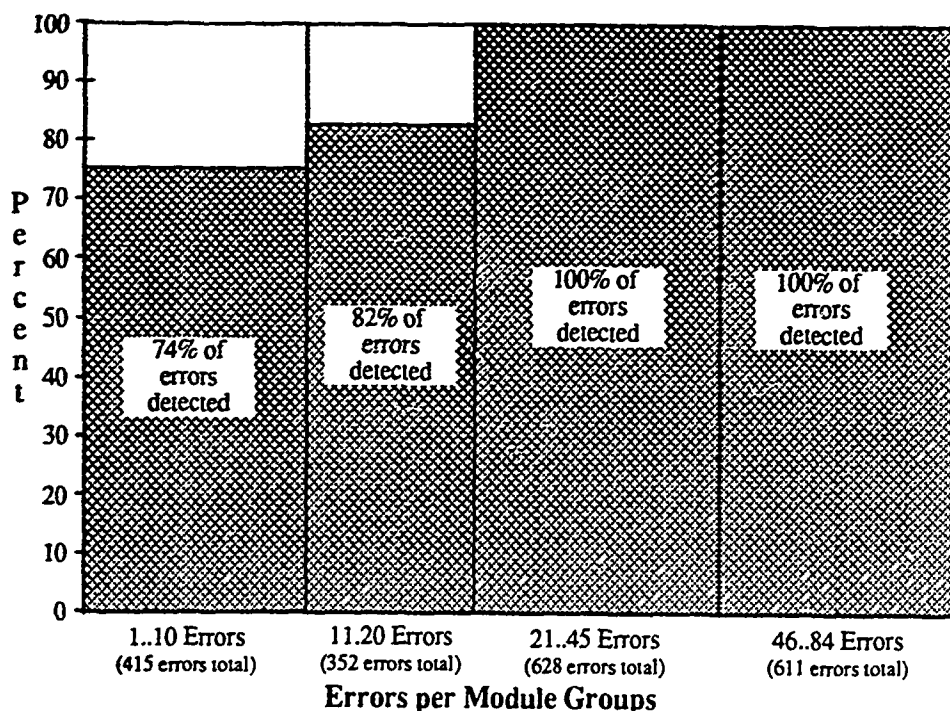


Figure 2: The Performance of D(G) on the STANFINS Test Bed

On the Rational, code can exist in three states; the Source, Installed and Coded states. Ada library units are developed with a language and context sensitive editor in what is called the Source state. In the Source state, a library unit is in a simple text form and cannot be referenced from other library units. After a library unit has been made syntactically and semantically correct (e.g., all references to external package elements are correct), it may be promoted to what is referred to as the Installed state. When promoted to the Installed state, a library unit is rendered into a DIANA representation. In the Installed state, a library unit can be referenced from other library units or analyzed via its DIANA representation. Most of the modules analyzed have been in the Installed state. In order to be executed, a library unit must be promoted again to the Coded state. The unit continues to be represented with DIANA structures, but gains other contextual information which is not important to the design metrics tool. Units in the Coded state may be analyzed, via their DIANA representation, exactly as in the Installed state. The DIANA representations of the library units in a system are bound together in a common structure which expresses the semantic structure of the system. This struc-

ture can be traversed via a DIANA interface package provided by Rational.

This structure and the interface packages were used as a basis for the analysis of Ada programs. By combining the defined packages with customized packages, an Ada design metrics analysis tool was developed. Our Design Metric Analyzer, or DMA, exploits Rational's DIANA interface package to determine several metrics which are fundamental to the design metrics calculations, including:

- Number of calls to and from superordinate or subordinate subprograms
- Number of parameters passed to and from a subprogram
- Number of external variables referenced and modified by a subprogram
- Number of calls to task entry points
- Number of complex data types (records, arrays, access types) accessed
- Number of central calls (subprogram and task entry calls)

Generally, computation of data flow and fan-in and fan-out metrics is simply a matter of

traversing the DIANA structure and adding the reported counts. However, a few Ada constructs required some fine-tuning.

- Generics and generic instantiations - Each reference to an instantiation of a generic subprogram is counted as a reference to the underlying generic subprogram. Metrics are then reported for the generic subprograms.
- Tasks and task entries - Each entry call into a task is treated as if the task is a subprogram being called with the entry's parameter list. Each of these calls adds to the data flow and fan-in metrics for the task itself according to the parameter list of the task entry. Metrics are then reported for the task.
- Separate subprograms - These subprograms are treated as though they are not separate.
- Renamed subprograms and variables - Metrics are applied to the original version of the subprogram or variable which was renamed.
- External variables - References are counted toward data-in and modifications are counted toward data-out.

Finally, the DMA calculates and reports the design metrics for each subprogram and for the system:

- D_e is calculated for each module from the data flow and fan-in and fan-out metrics. To avoid losing information in the calculation, the primitive metrics are assumed to have a minimum value of one.
- D_i is calculated for each module from the central call and complex data structure counts. Currently, input and output metrics are not included in the calculation due to the difficulty in identifying non-standard Ada I/O operations. (This count is added in other languages, such as C and Pascal, where standard I/O is easily recognized.)
- $D(G)$ is calculated by summing D_e and D_i .

To further support in the investigation of design metrics and also to provide information about the code itself, DMA also performs an analysis of the types of statements, declarations, other size measurements such as LOC, number of logical statements, number of comments, number of embedded comments, and many other primitive metrics.

The Practical Utility of $D(G)$ and Our Future Direction

We believe that the results obtained for $D(G)$ are very promising as one strives to analyze software designs. Once stress points are identified, the practitioner has the option of redesigning the system and then recalculating the metrics to check the redesign, or to leave the design as it is. There may be circumstances (such as the inherent complexity of the application or management's rationale for a given structure) where redesign is not the best alternative. However, even in these cases, once the potential trouble spots have been identified, the designer at least has the option to assign these difficult components to the most experienced developers or to simply allow more testing time for the resulting code.

All weighting factors are currently set to one. In our future work, these will be updated after we have collected enough data to determine their appropriate values. An effort is also underway to provide earlier support in the calculation of design metrics and to perform these calculations on a variety of platforms.

At this point, we see the need to evaluate the effectiveness of $D(G)$ as an indicator of software quality on more large-scale software systems. We have now begun that task. Only after sufficient evidence has been gathered will practitioners have confidence in using the design metric $D(G)$ in systems development.

References

1. Boehm, B. and P. Papaccio, "Understanding and Controlling Software Costs", *IEEE Transactions on Software Engineering*, Vol. SE-14, No. 10, pp. 1462--1477, October 1988.
2. Rombach, H.D., "Design Measurement: Some Lessons Learned", *IEEE Software*, Vol. 7, No. 2, pp.17--25, March 1990.

AUTOMATING TEST SYSTEMS FOR TACTICAL COMPUTERS

by Joyce Fowler, Bill Herleth, and Patrice Johnson

TELOS Systems Group
111 C. Street, Lawton, OK 73505

Abstract

TELOS has developed a method for resolving a growing need of the Department of Defense (DoD) to construct and maintain an automated test environment for combat systems. This paper focuses on some of the challenges presented in the development of the Test Support System (TSS) software: creation of a programming language dedicated to automated testing, conversion from the original version of TSS in FORTRAN to the current Ada version, porting the system between platforms, and selection of applicable software metrics to test code reliability and portability.

Introduction

The TSS was developed to provide an automated testing environment for the Fire Support System. When compared to manual testing, automated testing provides a more accurate, thorough, and cost effective means to test tactical system software, especially considering the requirement to perform regression testing when field system software is updated. The Test Support System has two phases, the translation phase, where an input file is processed for use by the system, and the driver phase, where the Tactical System Driver (TSD)

interacts in real time with the Tactical Equipment Under Test (TEUT).

Army tactical field computers have two ways of transmitting and receiving data, via the operator interface and through communication channels with other computers. The tactical system usually consists of a keyboard, video display terminal, and printer(s). TSS allows the host computer (VAX 11/785 with DMF-32 and DHU-11 adapter boards), to transmit data to the keyboard buffer, effectively mimicking the operator input, and to receive data from the printer port (see Figure 1). The same hardware connection is also used to return printer data from the TEUT to TSS for output and analysis, and also to provide feedback on the state of the TEUT to TSS. This feedback is used to synchronize the testing. The feedback status information requires a small amount of additional software code in the TEUT. Aside from the operator interface, a tactical field computer also receives and transmits data via its communications channels. The host computer for TSS can communicate with the field system via tactical modems built by TELOS. TSS is capable of testing up to four tactical field computers simultaneously, utilizing from one to seven communication channels.

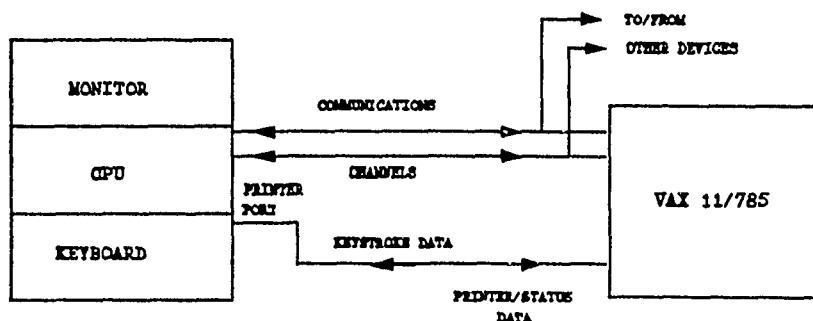


Figure 1. TSS Connection to Field System

In order to facilitate the delivery of keystroke and communication data to the TEUT, as well as synchronize the testing by having TSS wait for feedback from the TEUT, TSS uses a specially developed language that allows testers to write

program-like "scripts" for controlling the testing of a device. Table One lists some of the statements in the TSS language and provides a brief description of each.

Table 1. Some Test Support System Commands

<u>COMMAND</u>	<u>PARAMETERS</u>	<u>DESCRIPTION</u>
&&PROC	Procedure Name Classification MCAT Control FCSVER Version Baseband Channel X.25 Channel	There is one &&PROC statement per scenario. It defines various options used in the scenario both during translation and by the driver.
&&SUB	Device Name Subscriber Device Type Channel Number Address Unit Number Net Access Delay Version TSS Type Response	There is one &&SUB record for each channel for each real or simulated device in the scenario. It provides TSS with pertinent communication related information for the device channel.
&&KPI	Device Name Device Type Version	Each real printer port used in the scenario must have a &&KPI record. It provides TSS with pertinent keyboard related information on the device.
&&EDIT	Device Name Message Skeleton	&&EDIT is used to send keystroke data to a TEUT in order to fill out a message skeleton.
&&KEYS	Device Name Keystrokes	&&KEYS is used to send control keystroke data to a device (normally used for various "function" keys).
&&ENTE	Device Name Skeleton Segments	&&ENTE allows the user to specify only a subset of the fields in a edit skeleton; it is used to send keystroke data.
&&XMIT	Source Device Name Destination Name Message Skeleton	&&XMIT is used to send a prepackaged communications message to a device.
&&ENTX	Source Device Name Destination Name Skeleton Segments	&&ENTX allows the user to specify only a subset of a skeleton's field for transmission in a message.
&&FFM	Source Device Name Destination Name Message Type	&&FFM is used to send a free-form message to a device.
&&GDUI	Device Name	&&GDUI signals the existence of a Gun Display Unit Interface (GDUI).

<u>COMMAND</u>	<u>PARAMETERS</u>	<u>DESCRIPTION</u>
&&BCON	Message Type	Certain types of packets may be made of multiple messages. BCON appends one message to another so that multiple message packets can be transmitted from TSS to the TEUT.
&&RESP	Device Name Status	&&RESP changes the desired response of a network subscriber to that specified by status (acknowledge, negative acknowledge, or no response).
&&WALL	Time-out Device Name(s) Event(s)	&&WALL instructs the driver to wait for all of the specified conditions prior to proceeding.
&&WANY	Time-out Device Name(s) Event(s)	&&WANY instructs the driver to wait for any of the specified conditions prior to proceeding.
&&INCL	File Name	&&INCL causes an external file to be included in the translation of a scenario.
&&"MACRO"	P1,P2,...,P8	&&MACRO invokes a predefined macro.
&&TEXT	Text String	&&TEXT prints a text string to the driver video screen and output file when this record is processed by the driver.
&&STOP		&&STOP is the last statement in the scenario.

The first phase of TSS, translation, consists of three programs (see Figure 2) and some file maintenance utilities. The Tactical Message Editor performs syntax checking of the input file, expands macros, retrieves any included files, and builds data structures used in the translation process. Translator One (TR1) converts keystroke data and communication messages into the form required by the TEUT. Translator Two (TRN) converts the TSS statements into a form quickly and easily processed by the driver.

TSS was converted and redesigned in Ada for the following reasons:

- o The Department of Defense encourages, if not requires, the use of Ada as the programming language for military-related software.
- o Ease of portability using Ada.
- o The moderate size of the programs made them suitable for conversion.

The portability aspects - VAX to PC, are expected to enable a reduction in costs and provide the capability for a larger and more varied testing environment.

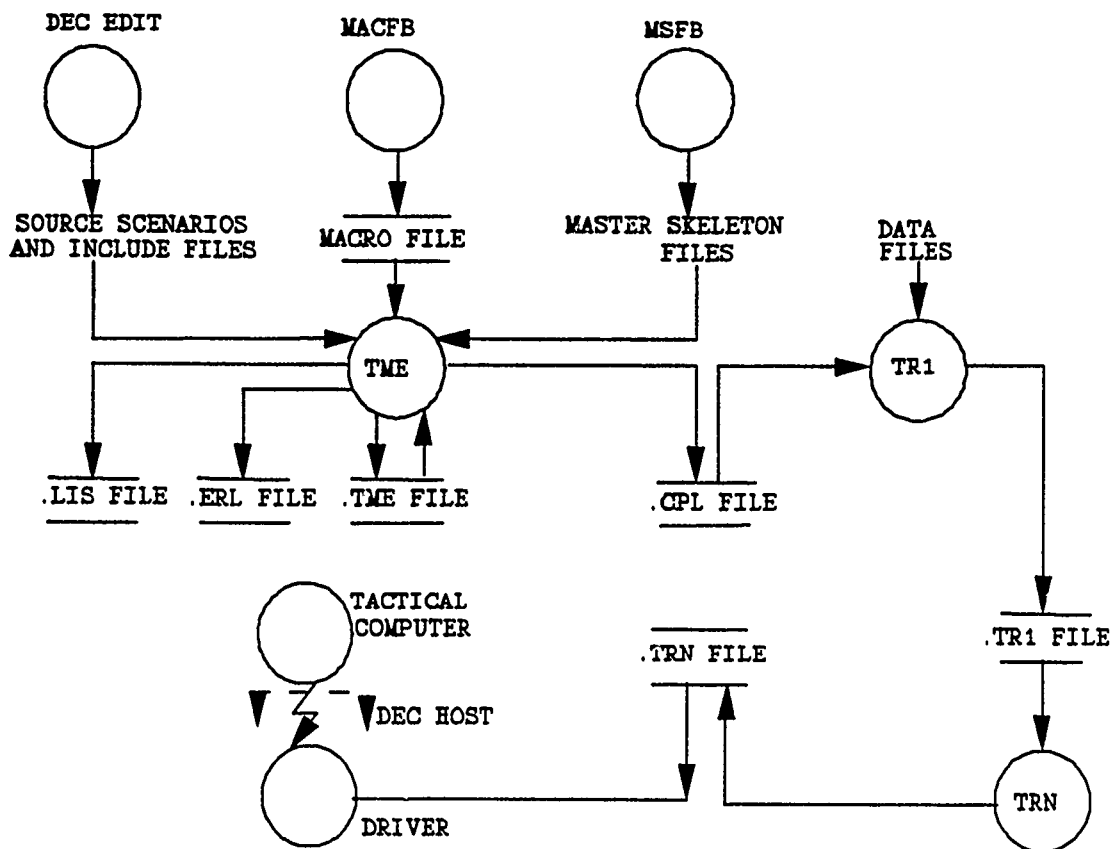


Figure 2. TSS Data Flow

TACTICAL MESSAGE EDITOR (TME)

The Tactical Message Editor (TME) performs many functions in the Test Support System including:

- o Syntax checking of the command language.
- o Processing of &&ENTE and &&ENTX commands, which allow the insertion of message information into predefined message skeletons.
- o Building and verifying the subscriber table from &&SUB, &&KPI, and &&GDUI records.
- o Expansion of message skeletons, and &&INCL records, which are external files that contain other TSS commands that will be included in the scenario, and, also, predefined "MACRO" commands, which can be any TSS command, except an &&INCL.
- o A capability to create and edit scenarios (logical constructs of TSS commands).
- o The creation of a translator file, which is the input file for (TR1).

An example of a scenario using the TSS commands would be as follows:

```
&&PROC 1.1 CLASS=UNC
&&SUB (CANNON,SB: / /C/AN/NON,N,1,N,,1,9,R,A)
&&SUB (DMD,SB: / / / /DMD,T,1,X,,1,9,S,N)
&&KPI (CANNON,N,9)
&&EDIT (CANNON,N/SYS;SETUP)
&&WALL (30,CANNON:EOS)
&&KEYS (CANNON) [ALARM ACK]
&&WALL (30,CANNON:EOS)
&&KEYS (CANNON) [RCVD MSG]
&&WALL (30,CANNON:EOS)
&&XMIT (DMD,CANNON,T/FRGRID)
&&WALL (30,CANNON:EOS)
&&KEYS (CANNON) [PRINT]
&&WALL (30,CANNON:EOS)
&&KEYS (CANNON) [EXEC]
&&WALL (30,CANNON:EOS)
&&KEYS (CANNON) [PREV SEG]
&&WALL (30,CANNON:EOS)
&&KEYS (CANNON) [PRINT]
&&WALL (30,CANNON:EOS)
&&STOP
```

For the records &&EDIT (CANNON,N/SYS;SETUP) and &&XMIT (DMD,CANNON,T/FRGRID), TME will display the following CANNON format of the SYS;SETUP skeleton and a FRGRID message format and allow the user to edit it:

SYS;SETUP MESSAGE

```
SYS;SETUP;C:      ;CALIBR;          ;MODEL;          ;
TIME: / / / ;DATE: / / / ;DRI: ;TGTNO:      /      ;
METUSE: ;METUPD: ;METDEL: ;MTOTMR;      ;
ATP: ;MPOLL: ;PRINT: / ;SCREEN: ;ATHFM; ;ATHDB; ;
CPHVTI: ;RPTAMO: / / / / ;FSOXMIT: / / / / ;
GDUPREAM:      ;BEEP: ?
```

FRGRID MESSAGE

```
FRGRID:      ;DIR:      /      ;CORD:      /      /      ;GRID:      ;
TYPE:      /      ;DOP:      ;SIZE:      /      /      ;ATT:      ;STR:      ;
SHELLFZ:      ;CONTROL:      ;ANGLE:      ;PRI: ;MIS: ;OBS: ;
TGT:      ;VOL: ?
```

The edited messages will then be put into the output file of TME.

The prior example scenario could be run using macros if the following macros had been defined.

```
&&BEGIN:A/R
&&KEYS (CANNON) [ALARM ACK]
&&WALL (30,CANNON:EOS)
&&KEYS (CANNON) [RCVD MSG]
&&WALL (30,CANNON:EOS)
&&END:A/R

&&BEGIN:P/E/PS
&&KEYS (CANNON) [PRINT]
&&WALL (30,CANNON:EOS)
&&KEYS (CANNON) [EXEC]
&&WALL (30,CANNON:EOS)
&&KEYS (CANNON) [PREV SEG]
&&WALL (30,CANNON:EOS)
&&END:P/E/PS
```

The scenario with macros would be entered as follows:

```
&&PROC 1.1 CLASS=UNC
&&SUB (CANNON,SB: // /C/AN/NON,N,1,N,,9,R,A)
&&SUB (DMD,SB: // // /DMD,T,1,X,,1,9,S,N)
&&KPI (CANNON,N,9)
&&EDIT (CANNON,N/SYS;SETUP)
&&WALL (30,CANNON:EOS)
&&A/R
&&XMIT (DMD,CANNON,T/FRGRID)
&&WALL (30,CANNON:EOS)
&&P/E/PS
&&KEYS (CANNON) [PRINT]
&&WALL (30,CANNON:EOS)
&&STOP
```

In the previous FORTRAN version of TME, a scenario was scanned twice during the processing of a scenario. The first pass expanded all &&INCL and "MACRO" commands, if the MACRO option was selected. For the conversion of TME in Ada, it was decided to have only one pass. When Ada TME encounters a &&INCL or a "MACRO" command, it opens the include file or macro, inserts all parameters for the macro, if any, and processes each command inside the scenario file. Since the FORTRAN TME wrote the output of the first pass to an intermediate file prior to the second pass, by eliminating one pass of the processing, the Ada TME reduces the number of disk accesses required to process a given scenario. The reduced I/O used in the Ada TME allows it to process scenarios faster than the FORTRAN counterpart. As Figure 3 shows, this effect is amplified when the host computer is heavily loaded, since an I/O bound job with many other processes competing for the CPU will wait to regain access to the processor, even after a disk operation is complete,

while the process will be rescheduled promptly upon finishing a disk operation on a lightly loaded system.

Other design considerations for the conversion of TME from FORTRAN to Ada included: (1) Grouping FORTRAN modules into logical Ada packages to reduce package dependencies; (2) Reusability of code by incorporating source from other programs already written in Ada such as the Variable Format Message Editor Device (VFMED), the Automated Ballistics Testing Capability System (ABTCS), and the Generic Target Acquisition Device (GTAD); and (3) Portability factors such as excluding System Library routines and the differences between the function keys on the VAX terminals and the PCs.

The FORTRAN TME has 5100 lines of source code compared to 15,000 in the converted Ada version. The additional Ada code consists of enhanced capabilities and code reused from various projects, including the VFMED message editor.

Some differences between FORTRAN and Ada that had to be considered were:

FORTRAN

1. Global structures.
2. In functions, there can be more than one parameter passed to the function being called.
3. In equating two string variables, they do not have to be of the same length.

Ada

1. Parameter passing replaced global structures.
2. There can only be one "out" parameter.
3. Two variable strings must be of the same length or the longer of the two string variables must have a range that equals the length of the smaller string variable.

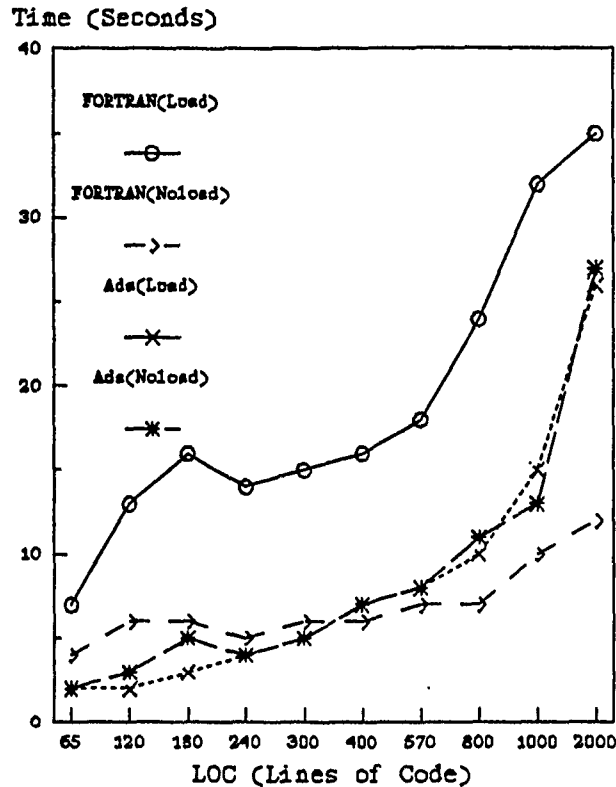


Figure 3. One Pass vs. Two Pass
Timing Comparisons

TRANSLATOR ONE

Translator One takes the output from TME and extracts keystroke data from edit message skeletons and compresses communication messages into their transmission byte stream. TME checks the scenario external to skeletons for syntax errors, TR1 may find additional errors in a scenario internal to a skeleton as it processes it for compression.

Keystroke processing in TR1 consists of extracting from a skeleton the actual keystroke data to send to the field equipment. While the keystroke extraction is occurring, TR1 makes certain optimizations to the byte stream for a more efficient delivery of data. These optimizations include substituting tabs for blank fields and down arrow keys for blank lines.

Communication message compression in TR1 consists of identifying the message by the message name and type and then converting each field in the message to its corresponding byte stream. The &&XMIT command is used to generate a communication message. Message data is entered into skeletons that are much like &&EDIT skeletons except that there are no protected fields. The message compression for a field may be as simple as copying the input data to the output for transmission as ASCII information or it may involve converting the input to a binary form prior to transmission. Often the data sent depends on input from multiple fields. There are four types of messages that are compressed: Bit-Oriented messages, fixed format messages, variable format messages, and Launcher messages. Each of the types of messages have unique characteristics that require them to be treated separately.

The message compression part of TR1 was completely redesigned when the program was converted from FORTRAN to Ada. The FORTRAN version coded the specific details of the compression into the software. Because the details of compression change with some regularity, this meant recompiling the code each time a change was made. The new Ada version keeps as many of the compression details as possible in various data files. With the data in files instead of in the code, it is possible to change many of the details of compressing a message without recompiling. Dynamically allocated data structures, using records with discriminants, store the message compression details. Redesigning the message compression in TR1 to use tables has reduced the number of lines of code in the program by a factor of two. The processing times for the FORTRAN and Ada versions are about the same for all but the smallest scenarios, when the time required to process the data files contributes significantly to the overall run time of the program.

ADAMAT

AdaMAT (Version 2.0), a code quality analyzer tool developed by the Dynamics Research Corporation (DRC), was used to test the TSS code converted from FORTRAN (Version 5.6) to Ada (Version 2.2). AdaMAT is a source code analyzer that reports on key Ada-specific quality metrics.

By scanning Ada source code to count the occurrences of specific programming practices of the Ada language, such as types, exceptions, and labels, Adamat produces a report (See Figure 4) indicating levels of quality and possible problem areas.

It uses a metrics framework developed by DRC for static analysis of compilable source code. The AdaMAT metrics framework is a hierarchically organized set of quality metrics (See Figure 6) that measure both adherence to recognized software quality principles and the inherent complexity of software. An example of how AdaMAT tallies scores of metrics is included in Figure 5.

DEV\$COPS2:[TSS.NEW.SOLD_OFF]UTILITY_PACKAGE.REP;1									
Score	Good	Total	Level	-----	Metric Name	Module	Source		
0.36	526	1473	1	-----	RELIABILITY	over modules	UTILITY		
0.47	1145	2437	1	-----	MAINTAINABILITY	over modules	UTILITY		
0.86	4129	4811	1	-----	PORTABILITY	over modules	UTILITY		
0.74	5021	6819	1	-----	ALL_FACTORS	over modules	UTILITY		
0.36	150	420	2	-----	ANOMALY_MANAGEMENT	over modules	UTILITY		
0.37	70	187	3	-----	PREVENTION	over modules	UTILITY		
0.28	31	109	4	-----	APPLICATIVE_DECLARAT	over modules	UTILITY		
	0	0	5	-----	APPLICATIVE_DECL_SPE	over modules	UTILITY		
0.28	31	109	4	-----	APPLICATIVE_DECL_BOD	over modules	UTILITY		
0.49	37	75	4	-----	DEFAULT_INITIALIZATI	over modules	UTILITY		

Figure 4. Example AdaMAT Report

```
EXAMPLE:  user_types_body
NON-ADHERENCE to metric
package body base_conversions is
  function base_10_to_2(number : in natural) return natural is
  --...
    result : natural := 0;
  begin
  --...
    return result;
  end base_conversions;

score(for the NON-ADHERENCE example above)user_types_body
0 good 1 bad 1 total
```

ADHERENCE to metric

```

package base_conversions is
  type binary is private;
  zero_base_2 : constant binary;
  type decimal is new natural;
  function base_10_to_2 (number : in decimal) return binary;
  ....
private
  max_bits : constant := 16;
  subtype binary_bits is character range 0 ... 1;
  type binary_range is range 0 .. max_bits-1;
  type binary is array(binary_range) of binary_bits;
  zero_base_2 : constant binary := binary'(others => 0);
end base_conversions;

package body base_conversions is
  function base_10_to_2(number : in decimal) return binary is
    temp : string(1 .. 2);
    copy : decimal := number;
    result : binary := binary_zero;
  begin
    for i in binary_range loop
      temp := decimal'image (copy mod 2);
      result(i) := temp(2);
      exit when copy <= 1;
      copy := copy / 2;
    end loop;
    return result;
  end base_10_to_2;
  ....
end base_conversions;

```

score (for the ADHERENCE example) user_types_body
 5 good 2 bad 7 total

Figure 5. AdaMAT Scores

A six step procedure was used in analyzing the converted TSS code:

- o Ran TSS packages through AdaMAT using all metrics with the output report in multiple package compare form. (See Figure 4 for AdaMAT report format example). This established an initial baseline of scores.
- o Removed a department-selected list of metrics from inclusion within the AdaMAT analysis and ran the report at the package level again. These metrics were considered inappropriate for application to the complex, multi-dependency software necessary for dealing with tactical equipment testing. (See Table 2 for a list of metrics that were removed). Note scores.
- o Generated a run through at the package level to show only the packages with any metrics below .25 (a very poor score suggesting a possible quality problem area). Noted applicable packages for the following more specific run.
- o Ran AdaMAT at the procedure/function level on the resultant packages that were revealed in the previous run to show which particular modules might be causing such poor quality. Fixed procedures/functions.
- o Ran again at the package level without the .25 restriction to see if suitable scores were obtained.

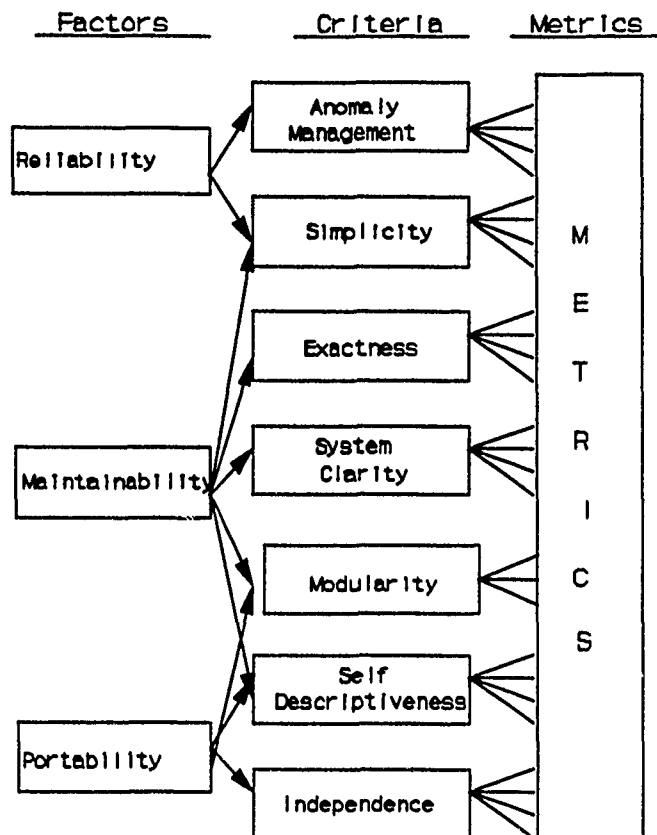


Figure 6. AdaMAT HIERARCHY

- o Since the TSS project was slated to port the code from a VAX 8810 to a UNIX based PC, a final run through AdaMAT at the procedure/function level was performed. The factor, Portability, depends upon the Criteria - Modularity, Self-Descriptiveness(Comments) and Independence(Machine/System Dependencies). The factors Self-Descriptiveness and Modularity were removed. It was felt that the major concern with porting the code was in Machine/System Dependencies. We noted scores for all procedures to identify possible future problems in porting between platforms.

Table 2. AdaMAT Non-applicable Deleted Metrics

<u>CALLS_TO_PROCEDURES</u>	<u>BRANCH_CONSTRUCTS</u>
<u>SINGLE_EXIT_SUBPROGRAM</u>	<u>FOR_LOOPS</u>
<u>LEVEL_OF_NESTING</u>	<u>STRUCTURED_BRANCH_CONSTRUCT</u>
<u>NON_BACK_BRANCH_CONSTRUCT</u>	<u>DECISIONS</u>
<u>BRANCH_AND_NESTING</u>	<u>NO_WHILE_LOOPS</u>

AdaMAT proved to be a useful tool for the TSS project in several ways:

- o Being able to predict at the procedure level, exactly which routines would need to be changed in order to port code between platforms, allowing for more comprehensive future planning and scheduling of manpower and resources.

- o By evaluating metric scores and making changes to improve them, programmers found they were subconsciously retraining themselves into using better Ada coding practices and, found that better scores and improved coding became almost routine after working with AdaMAT.
- o Having a tool that automatically reveals and measures potential code maintenance problems, is expected to save future man-hours for the department when TSS needs to be modified or errors corrected. Also, when certain levels of AdaMAT scores are eventually required by the department, a minimum standard of quality will be met.

As with any tool, careful consideration must be taken as to how to properly use AdaMAT. Human nature suggests that everyone wants to attain 100% scores. However, AdaMAT is a measuring tool that attempts to measure a very broad spectrum of data items. Not all items are necessarily applicable to all types of software. Is it reasonable to use the same metrics to measure quality for database software as would be used to measure a very involved scientific application? When used properly, AdaMAT can provide very helpful information and even reasonable standards, however; if required numeric scores for various metrics are mandated without proper study the resulting effort to attain these scores could be frustrating, time consuming and possibly produce less suitable code for the specific application.

CONCLUSION AND FUTURE WORK

Work is continuing on the TSS porting and conversion to Ada. TRN is scheduled to be rewritten in Ada and "folded into" TR1. The effort to port the TSD from the VMS operating system to UNIX should be underway by the time of the conference. Since the VMS driver utilizes asynchronous system traps and other operating systems services in its design, much of it will have to be reworked for UNIX.

In addition to converting TSS to a new language, new capabilities are being added to the system. Work is underway to add X.25 communications functionality to the system. There are also plans to design into TSS the capability to interact with computers having a graphical interface.

TELOS is working on various "intelligent" front ends to TSS that will allow more complete and faster testing of the software developed for tactical system computers. A new

version of the Automated Ballistics Test Capability System (ABTCS) is under development which will create ballistics scenarios from databases supplied by the Ballistic Research Laboratory. Generating and running ballistics scenarios automatically reduces the time it takes to a small fraction of that necessary to run the same tests manually. Another project under development will add the capability for the Fire Support Systems Interoperability Specifications (FSSIS) system to generate TSS scenarios. Generating scenarios directly from the specifications may provide a more complete test of the tactical field computers than is presently done utilizing human generated scripts.

The role of ADAMAT at TELOS is still evolving. Final decisions have yet to be made on what metrics to include when using the program and what scores are necessary for software to be considered acceptable.

REFERENCES

1. Dynamics Research Corporation, AdaMAT Version 2.0, Product Overview, An Introduction To The Concepts and Principles Of The Ada Measurement And Analysis Tool.
2. TELOS Systems Group, Agenda For The Forward Entry Device Met/Survey/Radar Software Specification Review, March 5, 1991.
3. TELOS Systems Group, Computer Program Development Specification For The Battery Computer Unit Operating System For the AN/GYK-29 Computer System, March 1, 1991.
4. TELOS Systems Group, Draft User's Manual For Test Support System For The Tactical Support System For The Tactical Computer Systems, Suspense date: December 1, 1992.

ABOUT THE AUTHORS

Joyce I. Fowler is a Systems Engineer at Telos Federal Systems with over ten years of experience in software development and engineering research. She has a diversified background which includes hardware and software design. She is currently in charge of a group working on automated software testing tools for army tactical equipment. Ms. Fowler received her B.S. degrees in Psychology and in Electrical Engineering from Washington University in St. Louis and is currently working on an MBA degree. Ms. Fowler is a member of SIGAda and DECUS.

Bill Herleth is a Software Engineer at Telos Federal Systems in Lawton, Oklahoma. He has a B.S. in Physics and M.S. in Computer Science from the University of Missouri-Columbia.

Patrice Johnson is a Software Programmer for the Facilities Management Section at Telos Federal Systems at Fort Sill, Oklahoma. She has a B.S. degree in Business Administration and a B.A. degree in Mathematics from Cameron University.

SO MUCH TO MEASURE – SO LITTLE TIME TO MEASURE IT The Need for Resource-Constrained Management Metrics Programs

**Stewart Fenick
U.S. Army CECOM
Fort Monmouth, NJ**

**Dr. Harry F. Joiner
Telos Corporation
Shrewsbury, NJ**

Abstract - Approaches to a resource-constrained management metrics program are described. They are based on the principle that not all data must be collected and analyzed at all data points. Resources can be conserved by utilizing an issue-driven, optional metrics selection scheme. Strategies are suggested for reducing the amount of data collected and analyzed. Criteria for decisions on which metrics to collect and when to analyze additional data are based on identification of the project objectives and primary risks to their achievement. A "gating" scheme is explored that allows the manager to look first at those metrics which are of highest priority and then at only those additional metrics that address any identified problem areas. The gating scheme and metrics hierarchy are described in terms of the initial set of executive management software metrics developed by the Software Engineering Directorate of the U.S. Army Communications-Electronics Command's Research, Development and Engineering Center.

Key words - Software metrics, project management

INTRODUCTION

There are numerous alternative management metrics sets recommended for use on software projects in order to control the cost, schedule and quality of those projects.^{1, 2, 3, 4, 5, 6, 7} However, there are a variety of reasons for taking a resource-constrained approach to the collection and analysis of software management metrics. Moderate sized projects, where management visibility and control are more easily accomplished, and organizations that are initiating a software management metrics program for the first time are examples that suggest need for a more conservative approach. Large projects with established metrics programs may also benefit from a resource-constrained strategy since it will not only reduce the man-hours required but may eliminate confusion for the manager caused by extraneous information. Techniques are suggested that can

reduce the amount of data that is collected and analyzed while minimizing the risk of undetected problems. They are founded on an issue-driven, optional metrics selection scheme.

One of the most common complaints by managers, when it comes to applying measurements to their software projects, is the drain on resources required to carry out the metrics program. Initiation of a metrics program normally requires:

- Additional training
- Specialized manpower
- Extended schedule and extra man-hours
- Added resources for tools and equipment

A second complaint is that measurement programs are viewed as intrusive in nature. They require collection of sensitive data; raise integrity, security and proprietary issues; and may provide a negative view of department or individual performance. Many are concerned that a measurement program will interfere with the progress of the project through added resource requirements and by negatively affecting how the work is performed. This last issue is a special problem that can only be dealt with when measurement becomes, as it should be, a part of a software project's infrastructure. In fact, measurement is necessary for achieving management control and process improvement.

Resource-constrained management metrics programs are an attempt to alleviate both of these complaints based on the premise that:

*It is not necessary to look at everything
all the time.*

Extraneous or incorrect metric data can obscure the useful data and lead to "in-corrective" action. By collection, analysis and reporting of only those measures that are directly or indirectly relevant to the "issues of the moment," measurement activity at specific data points can be restricted to only that which provides the most meaningful information. Thus, waste is eliminated, redundancy is greatly reduced, the confusion that can be caused by too much technical information is reduced, and results are presented in a manner that is more efficient and useful to executive managers. As a direct result, analyses needed at a particular data point are not only reduced in scope, but also in complexity. In addition, by not requiring that a continuous and probing evaluative process be applied, the fear and effects of intrusiveness are greatly alleviated.

Therefore, the foundation of a resource-constrained management metrics program is the same as that of any meaningful measurement program in practice today: issue-driven activities and decisions. At each data point, phase or activity, there is a relevant set of management metrics and indicators to look at in order to get a "big picture" snapshot of status, trends and potential problems of the moment. In effect, a tailorable set of measures is used for screening to determine if and when additional data should be analyzed and corrective action taken.

CECOM EXECUTIVE MANAGEMENT SOFTWARE METRICS

Based on this approach, the Software Engineering Directorate (SED) of the U.S. Army Communications-Electronics Command (CECOM) Research, Development and Engineering Center has developed a set of high-level management metrics that provide timely insight into the software development and support processes of large, complex, mission-critical software systems. The CECOM Executive Management Software Metrics (CEMSM)¹ includes 12 management metrics. The five resource metrics address the issues of cost, schedule, organizational capabilities, and computer resources. The four

progress measures expand on cost and schedule issues by assessing technical progress of CSU development, incremental releases, testing and program size. The three product quality metrics focus on design structure, requirements and design stability, fault profiles and fault densities and amount of rework.

The CEMSM set was derived from careful evaluation of the state of the practice and consists of the following:

1. Cost/Schedule Performance
2. CSU Development Progress
3. Design Structure
4. Host Computer Resource Utilization
5. Incremental Release
6. Requirements and Design Stability
7. Software Development Personnel
8. Software Fault Profile
9. Software Size
10. Staff Experience
11. Target Computer Resource Utilization
12. Test Progress

CEMSM are intended to be applied throughout the life cycle (see Figure 1). Data collection is done at timely data points, followed by analysis, intermetric correlations, and reporting. The CEMSM results consist of status assessments, trend indicators, deviation identification, and problem pointers. This quantitative information is combined with qualitative information from other sources (such as walk-throughs, inspections, and program reviews), as input to help managers determine when and what corrective actions are needed. Such an information-based decision process results in improved project awareness and control, and leads to improved processes and products. Monitoring of corrective action results leads not only to lessons learned for the current projects, but also to improved processes for use on new projects.

The CEMSM set is designed to apply to a wide variety of software development approaches. However, for some software processes, the details of the metrics should be adapted in order to better fit the process and reflect its status. Furthermore, some accommodation can be made to the internal capabilities of the contractor for data collection and reporting. This adaptation should

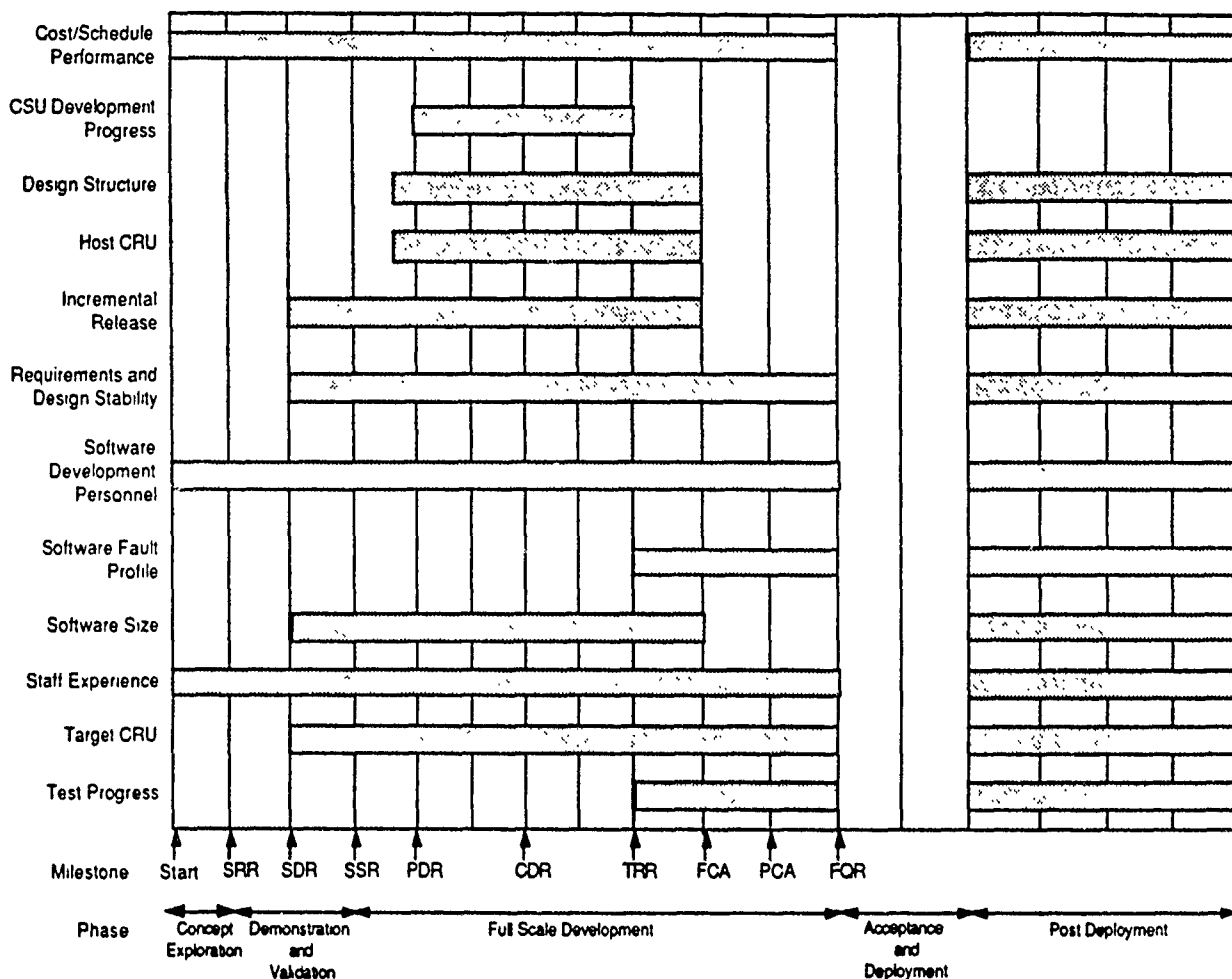


Figure 1. CEMSM Life Cycle Applicability

make the data more useful and help to establish support from the contractor by using data in a familiar format. The details of tailoring the CEMSM for object-oriented, spiral, evolutionary, and other development approaches is beyond the scope of this paper. Additional and/or substitute data may be required under those circumstances.

ADAPTATION TO CONSTRAINED RESOURCES

The CEMSM approach emphasizes a goal/risk-driven, flexible application strategy that allows for implementation and tailoring based on system specific concerns, issues and priorities and accommodates local measurement implementations regarding granularity, periodicity, and

partitioning. This approach provides several options in a resource constrained environment.

CEMSM forms a reasonable approach to management visibility and control for a large scale software development project. However, in a resource-constrained environment or for smaller scale projects, there are several effective techniques that can be used to reduce the cost and effort for collection and analysis of the metrics data. The following four approaches can be used separately or combined:

1. Some metrics may be collected in summary form, grouping information that would otherwise be broken out in a more detailed report

2. Only portions of some measures may be collected, reported or analyzed
3. Some metrics may be collected and/or reported less frequently
4. Some metrics may not be collected unless indications of problems occur

As an example of how these techniques can be applied, consider the metrics related to the area of staffing and level of effort. For a large project, these metrics would include collecting, reporting, and analyzing the following data on a monthly basis (for example, see the illustrative data in Figure 2):

- Planned and actual numbers of computer personnel working on the project by experience level and major activity
- Planned and actual personnel losses to each major activity within the project by experience level

Using the first approach in a constrained environment, the monthly reports might contain only the following summaries (see Figure 3):

- Planned and actual totals of computer personnel on the project
- Total actual losses of computer personnel from the project

KEY PERSONNEL														
ACTIVITY A			ACTIVITY B			ACTIVITY C			ACTIVITY D			TOTAL		
PLANNED	ACTUAL	LOSS	PLANNED	ACTUAL	LOSS	PLANNED	ACTUAL	LOSS	PLANNED	ACTUAL	LOSS	PLANNED	ACTUAL	LOSS
2	2	0	4	3	0	2	1	0	4	5	0	12	11	0
3	3	0	4	3	0	3	1	0	4	5	0	14	12	0
3	4	1	4	4	0	3	2	0	4	4	2	14	14	3
4	4	0	5	4	1	4	2	0	6	6	0	19	16	1
4	4	0	5	4	0	4	2	0	6	7	0	19	17	0
5	5	0	5	3	1	6	4	0	7	8	0	23	20	1
6	5	1	5	3	0	6	3	1	8	9	0	25	20	2
7	6	0	6	4	1	7	4	0	8	9	0	28	23	1
7			6			7			8			28		
7			6			7			7			27		
8			5			6			7			26		
														PERIOD
														1
														2
														3
														4
														5
														6
														7
														8
														9
														10
														11
														12

OTHER PERSONNEL														
ACTIVITY A			ACTIVITY B			ACTIVITY C			ACTIVITY D			TOTAL		
PLANNED	ACTUAL	LOSS	PLANNED	ACTUAL	LOSS	PLANNED	ACTUAL	LOSS	PLANNED	ACTUAL	LOSS	PLANNED	ACTUAL	LOSS
2	2	0	9	11	0	5	7	1	8	9	0	24	29	1
4	3	1	9	11	0	6	8	0	8	10	0	27	32	1
5	5	1	10	12	0	6	6	2	8	10	1	29	33	4
5	5	0	10	12	1	7	8	0	9	8	2	31	33	3
6	6	0	10	13	0	7	9	0	12	10	0	35	38	0
6	6	1	11	13	0	8	9	0	12	14	0	37	42	1
7	8	1	11	14	1	8	9	0	13	15	1	39	46	3
9	10	0	11	13	1	9	7	2	13	15	1	42	45	4
10			11			10			13			44		
10			9			10			12			41		
10			9			9			12			40		
10			9			9			12			40		
														PERIOD
														1
														2
														3
														4
														5
														6
														7
														8
														9
														10
														11
														12

TOTAL PERSONNEL														
ACTIVITY A			ACTIVITY B			ACTIVITY C			ACTIVITY D			TOTAL		
PLANNED	ACTUAL	LOSS	PLANNED	ACTUAL	LOSS	PLANNED	ACTUAL	LOSS	PLANNED	ACTUAL	LOSS	PLANNED	ACTUAL	LOSS
4	4	0	13	14	0	7	8	1	12	14	0	36	40	1
7	6	1	13	14	0	9	9	0	12	15	0	41	44	1
8	9	2	14	16	0	9	8	2	12	14	3	43	47	7
9	9	0	15	16	2	11	10	0	15	14	2	50	49	4
10	10	0	15	17	0	11	11	0	18	15	0	54	53	0
11	11	1	16	16	1	14	13	0	19	20	0	60	60	2
13	13	2	16	17	1	14	12	1	21	23	1	64	65	5
16	16	0	17	17	2	16	11	2	21	23	1	70	67	5
17			17			17			21			72		
17			15			17			19			68		
18			14			15			19			66		
18			14			15			19			66		
														PERIOD
														1
														2
														3
														4
														5
														6
														7
														8
														9
														10
														11
														12

Figure 2. Full-Scale Software Development Personnel Example

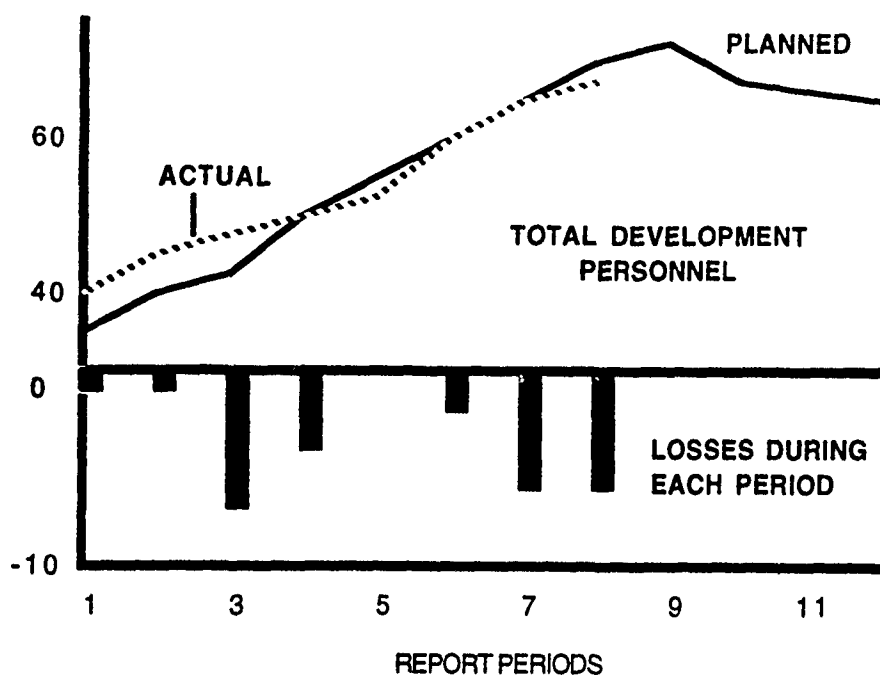


Figure 3. Software Development Total Personnel Example

The second approach might include reporting and analyzing only the planned and actual personnel losses to each major activity within the project by experience level or grade or only tracking the key personnel (see Figure 4).

The gating scheme discussed later is based on the third approach. The reports may be complete (for instance with monthly data points), but delivered less frequently (perhaps quarterly). For some projects or metrics, it is appropriate to stretch the time between data points to quarterly or longer periods of time.

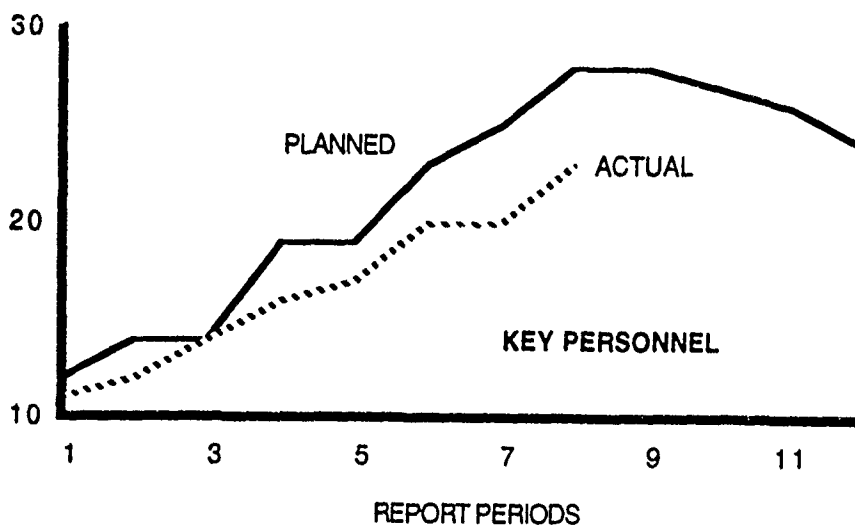


Figure 4. Software Development Key Personnel Example

The fourth approach was used by the Advanced Field Artillery Tactical Data System (AFATDS) Concept Evaluation project to track the level of effort on, and completion of, Software Development Folders (SDF). This reporting and analysis effort was initiated in order to monitor a particular concern: completing the SDFs for delivery at the end of the contract. See Figure 5⁸.

There are a variety of audiences involved in the use of metrics on a software project, and success of the metrics program depends upon their effective communication and efficient cooperation.

Managers should address the project from an organizational standpoint with a high-level perspective of the metrics information. On larger projects, they will normally be presented with a summary of the pertinent data and back-up information as requested. The software engineers and line supervisors will be expected to participate in the collection and authentication of the data as part of their role in the metrics program. Their view is from an operational or day to day perspective, frequently addressing only a limited portion of the data. The third audience, the metrics analyst, must serve as a bridge and synthesiz-

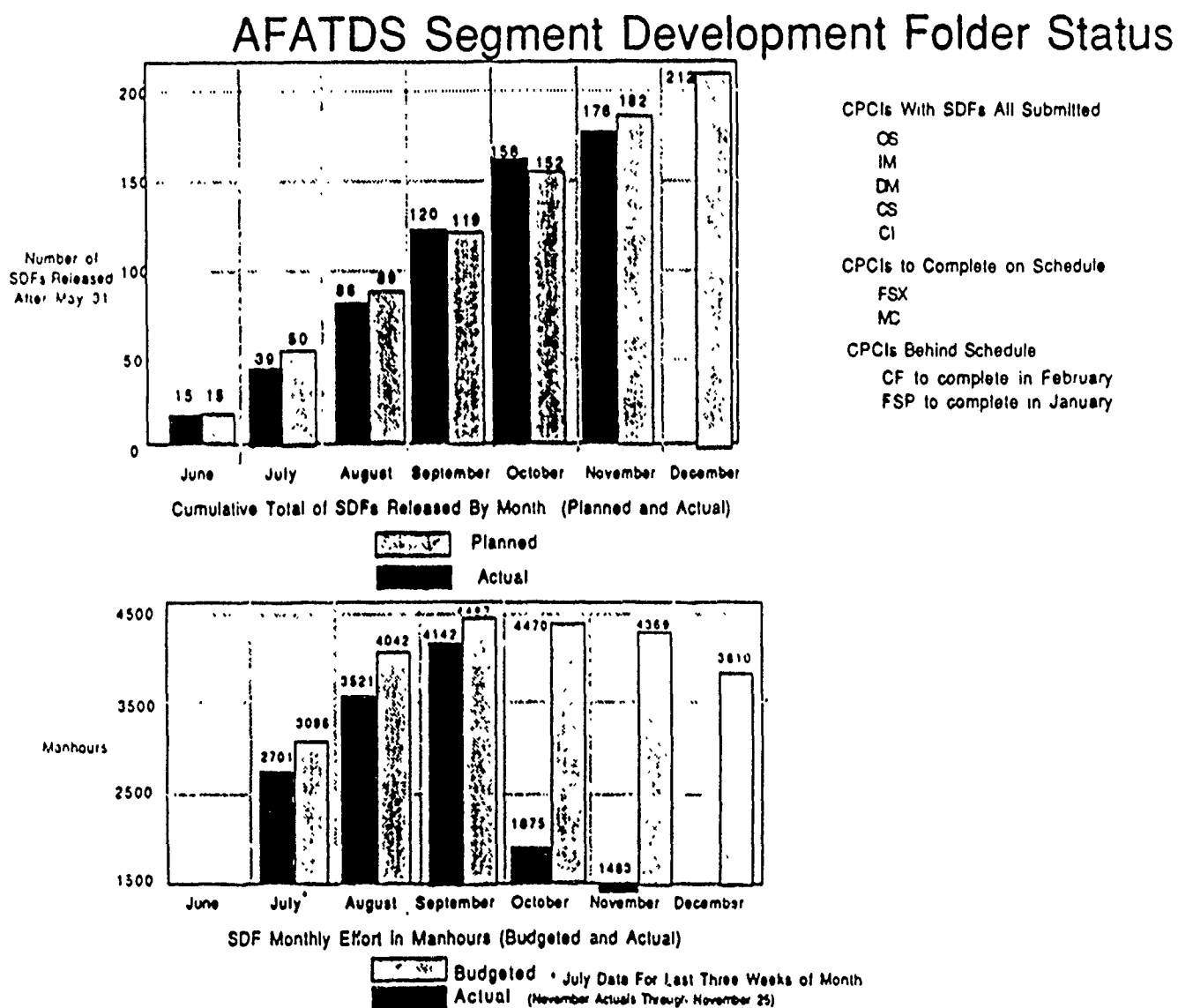


Figure 5. Example of the use of more detailed metrics information in order to monitor the level of effort on the completion of SDFs for AFATDS.

er in order to collect, analyze and present the data in a form that is most meaningful to each of the other two audiences. The analyst views the data as a whole, but with attention to the details that will generally be summarized for management review. In areas where potential problems lie or issues are being tracked, the manager will review the detailed information in order to fully understand the current status and trends. Particularly on smaller projects, there may be overlapping of these three roles in the same individual. However, it is necessary to understand the different data requirements, training objectives, and perspectives of the three audiences as part of the metrics tailoring process.

DECISION CRITERIA

The choice of metrics should be based on analysis of the project objectives, the program risks, and the need for communication and control.

One of the early decisions to make is how much data to collect. Each of the approaches saves on the total effort required by reducing the quantity of data to be analyzed whether the full set of data is collected or not. If the collection process is automated, the cost of gathering the data should be minimal and the data reports could be expanded later if needed for further analysis. However, when the data collection process itself is expensive, it may be important to reduce the amount of data collected. The principal disadvantage of reducing the amount of data collected is that it is not possible to do a more thorough analysis of the historical data from the early periods. The decision regarding collection should be made on a case by case, metric by metric basis.

The decision of which metrics to include in a resource-constrained program should be based on a complete understanding of the program objectives and potential risks. This approach extends the Basili-Rombach Goal-Question-Metric (GQM) paradigm⁹ and consists of the following steps:

1. Establish the goals - identify the critical project objectives
2. Develop questions that need to be answered in order to determine how well the

goals are being achieved and to pinpoint potential causes of risk

3. Define appropriate measures to answer the questions posed in the second step
4. Collect and analyze the metrics data defined in step 3.

The process may need to be repeated several times as the project moves from phase to phase and the conditions change. There are two guiding principles that must be understood and used:

The critical program objectives must be measured.

Metrics analysis must address the primary causes of risk to achieving those objectives.

In deciding which metrics to acquire and which to reduce or omit, the first step is to identify the project goals, followed immediately by identification of the most likely sources of risk to the program. For many years, cost and schedule (project goals) have been monitored without being able to control them. The successful projects have been able to identify, track and adjust for the likely causes of risk to the project, such as inadequate personnel, late delivery of needed equipment, etc. The CEMSM set addresses the most commonly encountered risk areas, but each project should determine which measures are critical, based on an assessment of the risk areas for that project and its circumstances. For instance, if a contractor has very limited experience with the Ada language or the application area, then it may be important to monitor both their training program and the availability of their critical experienced personnel. However, for contractors experienced in both the technology and application area, training and the availability of experienced personnel may be of little real concern to the project manager.

In assessing how many resources to apply to the measurement process, it is also important to remember that risk avoidance can represent very large dividends. The current emphasis on early problem identification is as important in project management as it is for software design issues.

Preventing a three month slip in the delivery date by early detection of inadequate host computer resources, could save tens of thousands of dollars on a small project and millions on a larger one. Is it worth the price of a focused metrics program? The answer is definitely yes.

MANAGEMENT INDICATOR GATES

Gates, or thresholds, can be used to trigger supplementary actions, including the collection and/or analysis of additional data. The gates should be based on the project objectives and historical data. The gating scheme is based on the relationship between different program areas and the resulting correlation between their respective metrics. The metrics that measure successful completion of high level project goals include Cost/Schedule Performance, Software Size, Test Progress and, perhaps, Design Structure and Target CRU (as quality measures). The second level measures address the causes of risk more directly than do the high level metrics and include such metrics as CSU Development Progress, Incremental Release, Requirements and Design Stability, and Software Fault Profile. The remaining measures address specific, though common, problem areas, such as Host CRU, Software Development Personnel, and Staff Experience. Additional metrics can be introduced to address system-specific concerns, such as Ada issues, reuse, productivity, etc. The hierarchy described here is intended to be flexible and is certainly not all inclusive. However, it does illustrate the distinctions between the goal oriented measures and their associated risk/cause oriented metrics.

There are two strategies to a resource-constrained gating scheme. The first involves the use of top level metrics at frequent intervals in order to determine the overall status of the project. These metrics address the major concerns and priorities of the specific system. The manager can then determine when, if, and which additional measurements should be taken and/or analyzed. This strategy is based on a hierarchy of metrics that provides increased visibility into the project operations.

Tracking cost, schedule and test results (the highest level goal metrics) until they show a slippage from planned levels is predicated on this first

strategy. It is the failure to take immediate, appropriate corrective action in the past that has so often proven fatal to success. The CEMSM set provides lower level metrics that can be used to help identify the reasons for such a slippage, such as requirements changes, inadequate personnel levels, lack of tools, etc., and to monitor the results of corrective action. A more appropriate level for most projects is to use some of the CEMSM set (CSU Development Progress, Incremental Release, Software Fault Profile, and Software Size) to track progress on the project through the development phases. As these measures vary from the planned values indicating a slippage in either schedule or functionality, additional data can be collected that will identify the cause of the slippage and indicate appropriate corrective actions. The results of the adjustments in the project should be tracked as long as the causes present a risk to the program objectives, i.e., if requirements stability is forcing a slip in the design program, then the Requirements Stability measure should be monitored for several months after the requirements appear to have stabilized.

The second strategy is similar to the first, but uses "back-up" or lower level metrics analyzed less frequently as a consistency check or early warning indicator. In other words, lower level metrics would be collected and reported, albeit less frequently, even if the top-level metrics did not indicate a deviation or potential problem. Since most problems are reflected in a trend analysis lasting over several months, it is unlikely that the metrics reflecting a problem area would surge to a critical level so quickly as to cause a crisis without an earlier indication. Furthermore, information from other sources, such as walk-throughs or discussions, will augment the early warning system by indicating any significant changes in status. Once early warnings are observed then the appropriate metrics can be analyzed more frequently, additional measures can be collected and reported, or both. This strategy is a trade-off between risk taking and resource saving. Any high risk areas should be measured with adequate frequency while the areas of low risk are monitored on a less frequent basis. The previous example would be modified under this approach by collecting and reviewing the lower level metrics (such as, CSU Development Progress, Incremen-

tal Release, Software Fault Profile, and Software Size) on a quarterly or semi-annual basis rather than monthly. There is far less risk when the lower level data is monitored periodically, even if only infrequently, than when the process is only triggered after indications of a problem.

Some gates in the reporting process may be automated on larger programs where significant amounts of data are collected and reported. Relying on fixed gate thresholds must be done with care. There is a trade-off between reducing the human resources devoted to the metrics analysis effort and losing the judgment that they represents. If the thresholds are used to trigger the collection or reporting of additional data, then a periodic check of the omitted data can reduce the risk that important information will not be considered in time to take appropriate action. In any case, it is generally preferable to collect and/or review more data rather than not enough.

REFERENCES

1. James N. McGhan and Peter B. Dyson, CECOM Executive Management Software Metrics (CEMSM) Guidebook, 31 October 1991.
2. Revised Implementation Guidelines for Software Management and Quality Indicators for AFATDS, Advanced Field Artillery Tactical Data Systems (AFATDS), July 1989.
3. Software Management Indicators, Management Insight, original U.S. Air Force Systems Command Pamphlet 800-43, January 1986, republished by Army Material Command, U.S. Department of the Army, AMC P 70-13, 1987.
4. Software Management Indicators, Management Quality Insight, original U.S. Air Force Systems Command Pamphlet 800-14, January 1987, republished by Army Material Command, U.S. Department of the Army, AMC P 70-14, 1987.
5. Software Management Indicators, U.S. Air Force Systems Command Pamphlet 800-43, August 1990.
6. STEP Metrics Initiatives Report, U.S. Army Software Test and Evaluation Panel, 25 March 1991
7. John H. Sintic and Harry F. Joiner, "Managing Software Quality" *Journal of Electronic Defense*, Vol. 12, No. 5, May 1989.
8. Harry F. Joiner and Stanley H. Levine, "Management Control through Software Metrics on a Large Ada Development Project - AFATDS" AFCEA Military/Government Computing Conference, January 1990.
9. V.R. Basili and H.D. Rombach, "The TAME Project: Toward Improvement-Oriented Software Environments" *IEEE Transactions on Software Engineering*, June 1988.

About the Authors:

Stewart Fenick
US Army, HQ CECOM
RDEC Sw Eng Directorate
AMSEL-RD-SE-ST-SE (Fenick)
Fort Monmouth, NJ 07703

Mr. Fenick serves as project leader of the CECOM Software Process Metrics Program. He has participated on various defense industry software metrics panels and working groups: DOD Software Technology for Reliable Systems (STARS) initiative; DOD SEI Metrics Initiative; RADC Software Quality Issues Working Group; The Technology Cooperative Program; and US Army Software Test and Evaluation Panel. He is currently Chairman of the CECOM Software Metrics Working Group. Mr. Fenick received a BEE from City College of New York.

Harry F. Joiner
Telos Systems Group
55 N. Gilbert Street
Shrewsbury, NJ 07702

Dr. Joiner is currently a Software Engineering Supervisor in charge of software metrics and reuse activities at Telos Systems Group Fort Monmouth Operations. His research interests include source code quality analysis, project management metrics, and software reuse as part of the engineering process. He is Vice Chair of the Reuse Working Group of the ACM Special Interest Group on Ada. Dr. Joiner received his BA from Texas Christian University and MS and PhD from Florida State University.

SOFTWARE PROCESS IMPROVEMENT PANEL

Moderator: Don O'Neill, Consultant

Panelists: Jonathon D. Addelston, PRC

James Dobbins, Defense Systems Management College

Stan Rifkin, Master Systems

Joan Weszka, IBM

PREPARING STUDENTS FOR INDUSTRY PANEL

Moderator: Dr. Genvieve Knight, Coppin State College

Panelists: Gary Ford, SEI

Dr. Joan S. Langdon, Bowie State University

Phyllis Villani, TRW Systems Division

SOFTWARE REUSE PANEL

Moderator: Dr. Harry F. Joiner, Telos Federal Systems

Panelists: Mr. William E. Carlson, Intermetrics

Dr. Kurt Fischer, OASD-C31

Stanley H. Levine, US Army

Dr. Charles Lillie, SAIC

An Implementation of a Generic Workstation Architecture for Command and Control Systems

by
Kirstan A. Vandersluis
SofTech, Inc.

and

Paul M. Richards
SofTech, Inc.

Abstract -- This paper reports on the software development of a Generic Workstation Architecture, developed in Ada, and its applicability to developing future command and control systems. A typical command and control system addressed by this architecture is a strategic missile or atmospheric warning command center system in which real-world situation information is managed by human operators. Generic architectures, in general, provide a portable and flexible framework for developing a system within a particular application domain. A Generic Workstation Architecture provides such a framework for the user interface component of a command and control system. This paper identifies the need for a Generic Workstation Architecture, and states the goals for this implementation that address the need. Highlights of the development effort including domain analysis, requirements analysis, design, and current implementation are discussed. Finally, the resulting system is evaluated against the stated goals, and future enhancement possibilities are presented. The use of Ada during the design and implementation is discussed where appropriate throughout the paper.

Index Terms -- Generic Architecture, Command and Control, Workstation, User Interface, Display.

1.0 INTRODUCTION

1.1 Background

There has been recent movement in the Department of Defense (DoD) towards establishing software reuse as the means for reducing future software system lifecycle costs. While these activities are not new, the current level of effort indicates a high degree of commitment not seen in the past. The Army

Information System Engineering Command has procured a software repository system called Reusable Ada Packages for Information Systems Development (RAPID), which has recently been embraced by the DoD Corporate Information Management program of the Defense Information Systems Agency [1]. The Air Force has procured the Central Archive for Reusable Defense Software (CARDS) system. The Air Force has also begun a program to implement generic architectures for all facets of Air Force command centers. This program is called Portable, Reusable, Integrated Software Modules (PRISM) [2]. These programs exemplify the commitment of the DoD to capitalize on software reuse to reduce the cost of both developing new systems and upgrading existing systems.

The concept of developing generic architectures for specific application areas is not new. Building on ideas from Dijkstra [3], Parnas [4], and others, Brown and Quanrud [5] have defined and discussed generic architectures. In their paper on "The Generic Architecture Approach to Reuseable Software", they define a generic architecture as providing a design together with a set of reuseable components. The design supports implementing the requirements of all applications in the chosen application domain. Brown and Quanrud conclude that generic architectures provide a high level of reuse among applications within a domain, which can lead to substantially lower development and maintenance costs than can be expected with other development approaches.

1.2 Overview

This paper describes the development of a Generic Workstation Architecture (GWA) which provides a reusable user interface component for modern command and control (C2) systems. Figure 1 shows a simplified architecture for a typical command and control system addressed by this paper. Sensors, other command centers, and external users exchange information with a command center. Information flows from external sources into the communications component within the command center. The information is transferred to a mission processing component for application-specific processing. Processed information is then relayed to the user interface component for display to operators. Operators also enter data into the user interface component for possible processing and transmittal to external users. It is this last component, the user interface, that the GWA addresses.

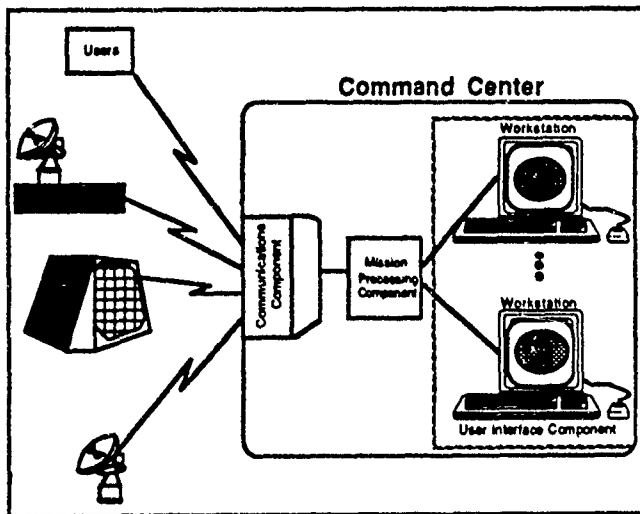


Figure 1. Command and Control System Architecture

The GWA implements the user interface component as an architecture, or software "platform", plus an extensible set of reusable software components. The architecture is seen as an executable framework that provides the necessary software support for two and three dimensional interactive graphics. The

architecture provides the resources to manage both data that is eventually transformed into visual form, and requests from the user to control the workstation environment. A system developer integrates the GWA into a system by interfacing to the GWA and extending the set of reusable components to suit the specific needs of the application. In this paper, we discuss the goals of the GWA project, its development, and current implementation.

Generic Workstation Architecture Goals

Several goals were defined for the GWA to address the need for a flexible, portable user interface environment. The goals are to achieve a high degree of software reuse in future C2 development efforts; to provide a rapid prototyping facility to aid in requirements definition; to provide an extensible workstation framework; to facilitate portation to multiple platforms; and to provide operational quality performance. The following paragraphs discuss these goals in detail. An evaluation of how well the GWA implementation satisfied these goals is provided in the Results section.

1.3.1 Software Reuse. Command and control system development is currently a multi-year, multi-million dollar activity due to the complexity of the problem and the difficulty of managing a large system development process. The DoD budget is apparently contracting yearly, without a corresponding decrease in need for software systems. One viable method for meeting software requirements within a reduced budget is to construct new systems using existing building blocks, rather than developing the entire system anew. Will Tracz notes increasing industry emphasis on reuse, quoting Barry Boehm, the director of the Software and Intelligence Systems Technology Office of the Defense Advanced Research Projects Agency (DARPA) [6]. Boehm has issued the "Megaprogramming Challenge" to researchers, encouraging the development of technology that allows software to be developed "one component at a time rather than one line of code at a time". The building block approach reduces overall complexity by hiding many implementation details within large scale components.

Using the GWA for developing new systems or upgrading existing systems is expected to significantly reduce the cost of development compared to developing a user interface component using other approaches. It is not expected that all requirements within the command and control application area can be implemented in a single system. Rather, the software architecture itself must define a mechanism for extensibility. The GWA must be tailorable by expanding or modifying existing classes of capabilities, or by adding new classes. In both cases, the GWA must provide the framework or high level design for new capabilities.

In order for the GWA to be reusable, interfaces to the surrounding environment must be well-defined. This environment includes the mission application, the database system, existing display software, and existing programs. The GWA must be structured to allow easy integration into a system. If an organization has existing display software, there should be a method to reuse this software in the GWA. If an organization has existing programs that satisfy system requirements, there should again be a method to integrate these programs into the GWA.

1.3.2 Rapid Prototyping. Often, a user has difficulty accurately expressing requirements until well into the development cycle. Grady Booch notes that requirements evolve during the development process as both users and developers gain a better understanding of the desired behavior of the system [7]. While users come to understand what the system is able to do, developers gain insight into the problem domain and ask better questions. This leads to a gradual convergence on the true desired behavior. Unfortunately, this process does not stop prior to intensive development work, but continues through the development cycle and throughout the life of the system. As a result, a deployed system often does not meet the users' true requirements. This malady is particularly acute in the DoD environment where personnel change positions frequently, especially in the context of a multi-year development effort.

One method for accelerating convergence on a system's desired behavior is to use a rapid prototyping technique. Both users and developers increase their knowledge about the true requirements by implementing a subset of the requirements, then operating the system composed of this subset. Prototyping the user interface, in contrast to other components, is particularly effective, since the user becomes familiar with his/her interactions with the system.

Rapid display prototyping allows users to evaluate and refine the static characteristics of displays early in the development process. Users can specify attributes, such as the general composition of display entities, with greater confidence that the finished system will match their requirements. The addition of a simulation capability provides a view of the displays under dynamic conditions, yielding a realistic operational simulation within which the users can evaluate and refine requirements.

1.3.3 Extensibility. Some systems we have analyzed express requirements to modify the system in the operational environment. In these systems, it is desirable to combine information that exists in the system in new ways dependent on current operational needs. The operator is given the capability to create and modify displays based on an existing set of displayable data. This capability is typically called "user defined displays". This type of flexibility in the deployed system could be highly desirable in situations where unanticipated "hot spots" develop in the real world, and existing displays do not provide the required graphical information. Rather than having to expend extensive resources for new displays in development, testing, and redeployment, a more flexible system would allow the user to create displays specific to the current situation. For example, if an unfriendly third world country developed a threatening ballistic missile capability, a missile warning system user might create a regional display of that country to allow close monitoring of missile launches.

1.3.4 Portability. It is highly desirable for the user interface implementation to be portable to

multiple workstation environments. This can be facilitated by basing the implementation on standard software tools. The use of Ada, the X Window system, PHIGS, and other standards will ease porting to other platforms. Environment specific features which do not adhere to standards must be isolated to specific software components, so that these components can be modified for new environments.

1.3.5 Performance. The generic nature of the GWA should not be permitted to significantly degrade the performance of the user interface component. The architecture and reuseable components must be operational quality, to allow integration with a high level of confidence in reliability and performance.

1.4 Approach

The development of the GWA consisted of a DoD-Standard-2167A compliant software development lifecycle consisting of Domain Analysis, Requirements Analysis, Design, Implementation, and Testing. The Requirements Phase was a two-step process of requirements specification and requirements analysis. Requirements were specified in a document similar to a System/Segment Specification (SSS). Requirements analysis was performed to develop software requirements documented in the Software Requirements Specification (SRS).

Requirements from several C2 System Specifications were examined as the basis for our GWA requirements. An architecture was built to support these, and other anticipated requirements. Our hope was that by empirically identifying and supporting user interface requirements from a number of systems, the architecture would support a broad scope of C2 systems.

The architecture was designed, implemented and tested using the Ada programming language. Ada was used as both a program design language and the implementation language. Its use has facilitated maintenance also, since we have a rich pool of personnel trained in Ada software development.

2.0 THE GWA DEVELOPMENT EFFORT

2.1 Domain Analysis

The requirements specification phase employed a form of domain analysis to bound the scope of the GWA. The objective for the GWA was to build a flexible architecture general enough to support the entire command and control application area. In order to achieve this goal, specifications for a number of Air Force strategic command and control systems were examined. The system specifications analyzed included MCCS/MSS [8], Granite Sentry [9], ASAT/BM/C3 [10], JSIC [11], AMHS [12] and CCA [13].

Domain analysis was performed to establish the relevance and type of each requirement contained in the various system specifications. Since the GWA focuses solely on the workstation processing, domain analysis concentrated primarily on the user interface requirements. However, system wide requirements such as data management, communications, message handling and security were also considered.

Each requirement was extracted and categorized into one of three basic domains: core requirements, representative requirements and system specific requirements. Core requirements are the common requirements found in most or all of the system specifications that would conceivably be needed by any command and control system. Representative requirements were those found in multiple systems that are typical of operations desired for command and control systems, but might not be needed to support any particular C2 system. System specific requirements are the esoteric requirements specific to a single system that are atypical of command and control systems. Figure 2 shows how the three categories are related.

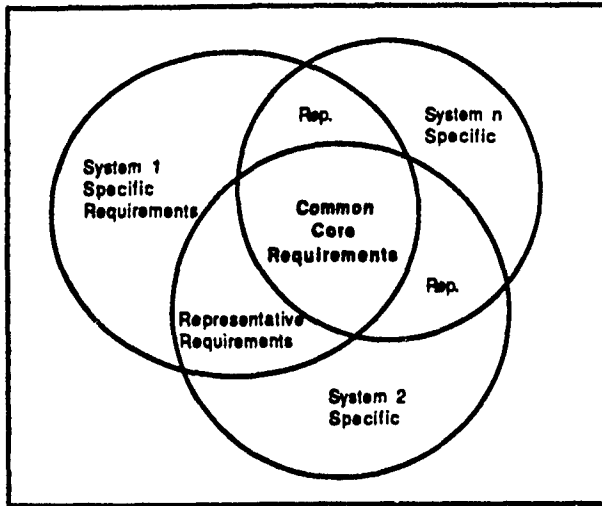


Figure 2. Requirements Categories

We captured the system level requirements in a document called the System/Segment Specification Insert (SSSI). This document follows the DoD-Standard-2167A SSS format and facilitates inclusion in the specification of a command center which incorporates the GWA for workstation processing. This is consistent with the goal of packaging the GWA for reuse; the documentation should be as reusable as the software. A system developer can extract appropriate portions from the SSSI for integration into his/her system level specification.

In addition to identifying actual command and control system requirements, we recognized the need to address the needs of future system developers. Based on our experience with command and control system development, we projected ourselves as system developers for future systems, and identified important features we would want to see in a reusable user interface component. These were:

- o Reliability,
- o Performance,
- o Portability,
- o Flexibility to add and delete capabilities,
- o Ease of integrating existing software components,
- o Ease of integrating existing programs

- o Ease of interfacing to the mission processing component, and
- o Ease of interfacing to the database system.

These items were defined as system constraints to be addressed during the design.

2.2 Requirements Analysis

During the Requirements analysis activity, the system level requirements defined in the SSSI were modeled in the Integrated Computer Aided Manufacturing (ICAM) Definition language (IDEF) to specify the operational requirements for the software. The IDEF language leads to an engineering model that presents all of the system requirements as operations with the data that is input or output from each operation and the important controls and mechanisms guiding each operation.

The specification of requirements in this form facilitates identification of derived requirements, such as recovery, that must be specified for a complete system. The specification of requirements in the IDEF model also provides an easily understood set of requirements that allows the designers to smoothly transition to a software design.

The software requirements were documented in an SRS which consisted largely of the IDEF model. The SRS also included several Quantitative Performance Requirements (QPRs), or constraints, that are typical of command and control systems, such as the time it would take to render a display after a user request.

2.3 Design

This section discusses the design of the GWA, briefly examining first the design approach, then the design itself. We emphasize those features in the design that promote the flexible and reusable nature of the GWA.

2.3.1 Design Approach. The design of the GWA followed a methodology derived from other projects at SofTech. The design was carried forward in two major phases: a process model definition phase and an object oriented phase. The two phases were somewhat interrelated;

although they proceeded sequentially, some iteration and refinement was required.

Before proceeding with the top level design we continued the domain analysis started in the requirements specification phase. Each of the SRS requirements was categorized as affecting one of two design elements: architectural considerations or reusable software components. The goal was to design an architecture, or software "platform", that could support all of the requirements by allowing components to be developed and "plugged-in" to the platform. The process model definition phase addressed the architectural requirements, while the object oriented phase emphasized the reusable component requirements.

In the process model definition phase, the functional activities defined in the requirements model were allocated to processes, and the interfaces between the processes were defined. These processes were later mapped to operating system processes. Issues such as performance, reliability, ease of integration with COTS products, and others were analyzed to assist in the definition of processes. The result of the process model phase was a set of programs defined using Ada as a program design language (PDL), and Ada packages representing the interfaces between the programs. The interface packages encapsulated the data passed between programs as Ada type definitions, and provided operations to send and receive the data.

In the object oriented phase, major object classes were defined for the software system. Object classes were defined based on entities extracted from the requirements model, and also based on the experience of the engineers on what abstractions were useful on previous projects. The result of the object oriented phase was a set of software objects representing either real-world or implementation-required entities, and the possible operations on these entities. The software objects were implemented using Ada to define the required data types and operations for the entities.

The final activity of the design phase was to combine the process model and object oriented model into a compilable Ada system. Integrating the models at this point served to verify the interfaces between components (by compiling these interfaces), and provided a baseline from which to start the coding phase.

The development of the GWA proceeded with an iterative build approach. The software requirements were partitioned into sets that could be scheduled for sequential implementation. The benefit of this approach is that it allows for the analysis, testing, and revision of the design at defined milestones (at the end of each build). At the same time, an initial operational capability was defined to include important features to make the GWA a viable, ready to integrate user interface component. The capabilities included in this initial operational capability build are described later in this paper in the section entitled Implementation.

2.3.2 Design Features. In this section, we discuss important features in the design that promote flexibility and reusability of the GWA. We first discuss the nature of a display within the GWA, then the interfaces to other components within a command and control system.

2.3.2.1 Displays. The object oriented design activity led to the important recognition that C2 displays are generally a composition of lower level display elements. This is not a new revelation, as graphics languages and systems have evolved towards a hierarchical, object-oriented approach in implementing graphical representations. Features such as display structures in GKS and PHIGS, and widgets and their counterparts in windowing systems such as Motif, DECwindows, Macintosh, and others, all follow the theme of defining elemental display objects from which to build higher level graphical entities. In the GWA, we have built a higher level of abstraction that can be viewed as a toolkit for C2 user interfaces. In the GWA, these display structures are called display elements.

A display element is any graphical entity that can be displayed on the active region of the workstation screen. This generally excludes the static banner area consisting of the title section, classification, date-time field, and menu area, although the GWA supports these elements as well. Display elements are classified in such a way as to group graphical entities with similar attributes. Examples of display element classes include:

Icon - graphical entity representing a real-world entity.

Table - matrix of information, arranged in rows and columns. Cells typically contain alpha-numeric data.

Alarm - graphical and/or audible entity intended to capture the attention of the operator.

Form - grouping of information, usually formatted to enhance understandability, and often used as a template for the user to enter information.

Overlay - logical grouping of other display elements that is identified by name, and that can be referenced and included in a display as a complete set.

Background - graphical diagram upon which other display elements are displayed. This is usually a type of map.

Text String - a character string.

Within each display element class, any number of display element types can be defined. For example, within the Icon class, we have defined ships, submarines, missiles, aircraft, satellites, and others. It is expected that most C2 display requirements will fit into these classes. However, the GWA does not limit either the number of classes or the number of display element types within a class.

Two categories of display elements are defined: static and dynamic. Static display elements are those whose appearance does not

change for the life of the display. An example of this would be a background for a display showing a fixed area on the globe. Dynamic display elements, on the other hand, rely on information outside the scope of the GWA to specify their appearance and/or location on the display. An example of a dynamic display element is an icon whose location and other attributes (color for example) are derived from data received from the mission application section of the C2 system. Dynamic display elements are much more complex since the GWA must map the data received from the application to the correct display element, and set the display element attributes prior to rendering. Correlating a data element to a display element involves mapping a tag from the received data to a tag in the display element. Establishing the attributes for a display element involves transforming the application data into graphical form. For example, the application may define the orientation of an aircraft - whether the aircraft is friendly or hostile. This orientation can then be translated to an icon color, possibly red for a hostile aircraft icon, and blue for one that is friendly. The mapping of data values to display attributes is isolated at the lowest level in the display element hierarchy. It is at the point where the display element is actually rendered that we map the data to visual attributes.

It is often necessary for a display system to perform some action as a result of a user operation. To implement this requirement, we defined the general capability to assign an action to a display element. A typical action might be to display a table of detailed information in response to the user clicking the mouse pointer on an icon.

We then defined a display in terms of its elemental characteristics. A display is defined as a group of display parameters (display title, menu option string, type of display - X or PHIGS, others), a list of static display elements, a list of dynamic or data driven display elements, and a list of actions. Taking this a step further, the information can be specified in a parameter file and loaded dynamically through a GWA menu option. The

parameter file is called a Display Description File (DDF).

The concept of displays was extended to include the definition of stand-alone applications. Existing applications can be integrated into the GWA by creating a special kind of DDF. The DDF specifies the name of the program, the name of the menu, and the menu command string that starts the application. The application is run in a window within the GWA environment.

The essential graphics processing capability of the GWA was implemented as a software component called a "display engine". This component is a virtual processing machine that accepts drawing commands and data for the supported display elements and performs the drawing operation, as shown in figure 3. The display engine loads the DDF into an internal data structure, usually at system initialization time. When the display engine receives a draw command with associated data from the mission processor, it scans the list of display structures, searching for a display element that matches the received command. The display element is drawn on all displays that have specified it.

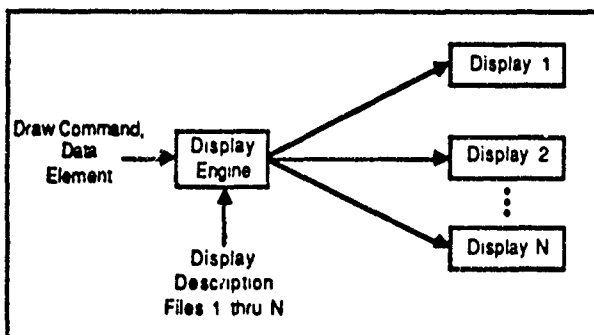


Figure 3. Display Engine

Figure 4 shows a simplified example. The DDF specifies a map background of the United States, a missile icon, a nuclear detonation icon, and an action that will display a table when the missile icon is activated by the user. The DDF is loaded into a memory-resident structure at initialization time. When the display engine

receives a draw command for a missile, for example, it scans its internal display list and finds that this display has specified a missile icon. The display engine renders the missile according to the data provided with the draw command, and stores the missile data for later use. If the user activates a missile icon by selecting it (clicking the mouse pointer on it), the display engine correlates the selected icon with the action defined for it, extracts the data previously received from the mission processor and performs the action. In this case, a table providing detailed missile information would be displayed.

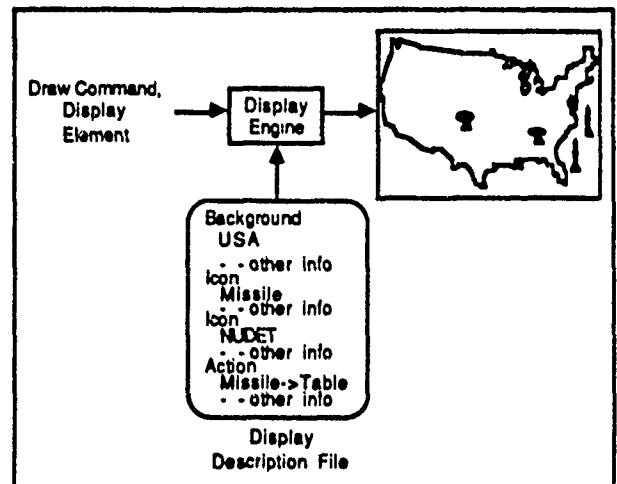


Figure 4. Display Engine Example

2.3.2.2 Interfaces. The interface to the mission processing segment is the most critical external interface for the GWA, since it is the source for all displayable data, and the destination for most user generated information. The primary concern for designing this interface was reliability of transferring the data. The mechanism used to ensure reliability of the interface was the strong typing provided by the Ada compiler. Interfaces were defined using Ada type definitions within a package that also defined the send and receive operations to be used by the mission processor and GWA. The interface package encapsulated the interface, and allowed the Ada compiler to verify the data passed on the interface.

The GWA uses a database capability to store data received from the mission application segment, and forward it to the software component that will render it. The GWA currently uses shared memory to store this information for performance reasons. The software components that implement the shared memory database can be replaced with components utilizing other data storage methods.

2.4 Implementation

This section describes the current implementation of the GWA, including specific display elements, architectural features, and support tools that have been developed as of this writing.

The current implementation of the GWA concentrates on a flexible software platform that is integrated with a basic set of display components. Many of the core requirements, some of the representative requirements and a few of the specific requirements are implemented. In addition, tools have been produced that allow a developer to rapidly construct, modify, and test displays without code modifications.

2.4.1 Software Architecture. The software architecture, or platform, is the executable framework that supports the basic processing capabilities of the system. This includes interfacing with external components such as the mission processor, receiving, translating, and storing display data, processing user interface requests, and self-contained system monitoring. Each of these types of processing are isolated in the implementation so that they can be easily modified, upgraded, or replaced. For example, a data management component is provided to store and retrieve dynamic display data. The data management component currently uses shared memory for faster performance, but could easily be replaced with a commercial Database Management System (DBMS) or other data storage mechanism.

The GWA includes monitoring functions to achieve the goal of reliability. The monitor is able to gather status information, report status (in the form of a message to the mission

processor), and restart independent software tasks that have failed without impact to the rest of the system.

Each DDF defines all parameters associated with a display. These include the desired background, a set of display elements, and actions associated with the display elements. The display engine processes the DDF at initialization, by loading the information into an internal data structure. The display engine then renders the display elements on the screen as defined by the DDFs. A display can be modified by modifying its DDF. A new display is created by creating a new DDF. A list of displays associated with the GWA is maintained as a separate file.

A special type of DDF, called an application DDF, allows a developer to specify an application program that executes on the target hardware as an integrated element of the GWA. This allows users of the GWA to incorporate existing software programs into the GWA system.

2.4.2 Display Elements. The display element classes and types currently defined include:

Icons:

- Aircraft
- Missile
- Submarine
- Ship
- Satellite
- Nuclear Detonation
- Radar

Tables:

- Missile Summary
- Nuclear Detonation Summary
- Missile
- Aircraft
- Satellite

Overlays:

- Arbitrary grouping of icons

Backgrounds:

- Arbitrary view of the globe
- *Encapsulated Displays

* Encapsulated displays are existing display software encapsulated as a software component, and integrated with the GWA as a complete display. This allows integration of displays that are already completely implemented.

2.4.3 Other Reusable Components. In addition to the reusable components provided to implement the display elements, the GWA provides a set of reusable software components as part of the GWA toolkit. Unlike display elements, these components are integrated into the GWA system as compiled library units. These include components to satisfy several common needs of C2 systems. These components include alarm rendering capabilities for both a panel of multiple alarms (Audio, Visual, or both), or pop-up alarms (Audio, Visual, or both), menu processing, and a static banner component which includes a system title, display title, data labels, and a clock. Each of these components can be replaced or tailored to meet the needs of a specific C2 system.

In the future, these commonly used components will be integrated with the display engine as additional display elements. This will allow a developer to specify the component in a manner similar to display elements, reducing the need for code changes for workstation modifications.

2.4.4 Tools. Two important tools were created to help developers implement a command and control system: the Display Builder and the Simulator. The Display Builder is a tool that allows a developer to interactively create a new display or modify an existing display interactively. The Display Builder allows the developer to select the background, static display elements and dynamic display elements along with associated actions. It also allows the developer to specify other parameters for the display, such as the display title, parent menu, and option that activates the display. The Display Builder stores the appropriate information in a DDF and optionally adds the new display to the display list so the display remains in the GWA configuration.

The Display Builder is integrated with the GWA as an application, and is therefore

available during execution of the GWA. This feature allows a user of a command and control system developed with the GWA to dynamically create and modify displays within the operational environment.

The Simulator is a tool that interfaces with the GWA and behaves as a mission processor. The Simulator allows a user to input individual data items or scenario files into the GWA, which are received, processed, and rendered by the display engine. The Simulator can also inject random sequences of data into the GWA. As with the case of the Display Builder, the Simulator can be integrated into the GWA by means of an application DDF. This is the typical configuration during display prototyping.

3.0 RESULTS

This section discusses the results of the GWA development project. Some general observations are offered, a subjective evaluation of how well the stated goals were satisfied is presented, then an overview of how the GWA would be used in the development of a command and control system is provided.

3.1 General

The use of a file-based display definition design greatly increases the flexibility for defining displays, not only for the developer, but as an interactive user capability as well. It is intuitive that the parsing and processing of display files and data structures adds to the overhead of the display rendering as compared to a hardcoded approach. However, the rapid advancements in workstation hardware processing capabilities dwarfs the relative overhead (primarily at display initialization time) of the GWA design approach. The GWA also supports the development of displays that can be initialized at system start-up or as the display is requested by the user, allowing designers to tune the system for faster response to critical displays.

The availability of development software and tools provides an easy method for rapid prototyping with several benefits. First, the displays can be rapidly constructed with the tools currently provided to stabilize the

requirements for a system under development. Secondly, the development phase for the graphics software is nearly completed once the prototype is completed, since displays are completely defined by the DDF constructed during the prototyping activity. Third, the prototypes are identical to the developed software in look-and-feel, which is not always the case with special prototyping tools. Finally, the prototype can be used as a simulation tool to ascertain the behavior and performance of the workstation software.

One current limitation of the GWA is the mixing and matching of display elements with one another. For example, some elements mix well (icons on backgrounds, tables on backgrounds), while others do not (X background with PHIGS foreground). This problem can be mitigated with the incorporation of additional display components such as map backgrounds developed in X for compatibility with other X display components. Another approach is to convert to a single graphics environment such as X. Although this may involve more effort in the development of reusable components (since X has no built-in 3 dimensional features), it provides for a more easily integrated, homogeneous system. It is likely that some system developers using the GWA will choose not to use PHIGS display elements because of performance. This does not, however, make the PHIGS display element features of the GWA less attractive. In fact, we see the ability to choose between single and multiple graphical systems as a benefit not found in other systems.

3.2 Evaluation

The following paragraphs offer an evaluation of goals stated in section 1.2 of this paper. The evaluations are subjective for the most part. A thorough, accurate, and quantitative evaluation can only be conducted after actually using the GWA in a development effort.

3.2.1 Software Reuse. The GWA supports reuse to the extent that the GWA can be used as the basis for the entire user interface for a command and control system. This is a much

higher level of reuse than can typically be expected from a reusable component library.

One major issue affecting the possible reuse of the GWA is the definition of its external interfaces. Matching the interfaces of the GWA to the other components in a command and control system is a fairly time-consuming task. Fortunately, data passing between the mission application and the GWA is defined by Ada type definitions which are isolated to a small number of packages. Though establishing this interface potentially involves defining a large number of Ada types, the approach provides a reliable interface mechanism since the Ada compiler verifies the data types flowing across the interface. The interface into the database system is also isolated to a small number of Ada packages. Using a different database system involves replacing these packages.

3.2.2 Rapid Prototyping. The GWA can be used to rapidly construct displays to assist users in analyzing and evaluating displays. Also, display elements that may be required for a new display can be developed and integrated into the GWA. Locations for new display elements are identified in the software, reducing the need for an extensive software re-design effort.

3.2.3 Extensibility. The GWA is extensible without coding changes in that displays can be created and modified within the parameters of the existing display element set. These changes can be made without modifying the source code. The display element set itself can be extended by integrating new display element code to the GWA. Again, the locations for adding new display element code are already defined, so only a small amount of design work is required. Once new display elements are defined, they can be used in virtually any combination with other display elements.

3.2.4 Portability. The GWA is currently implemented in Ada on a VAX/VMS system, using PHIGS, X, and DECwindows. Ada, PHIGS, and X are easily ported to other platforms. Both VMS and DECwindows are non-portable, so every effort was taken to isolate these interfaces to specific software components. In this way, when the system is ported, only these software

components need to be modified. Our current plan is to port the system to a fully standards based implementation. This implementation would use Ada, POSIX, X, and Open Software Foundation's Motif. This will ensure easy portation to any environment that supports these standards. These environments include workstations from Sun, Hewlett-Packard, IBM, and Digital Equipment Corporation.

3.2.5 Performance. Preliminary performance evaluation indicates that the GWA provides adequate performance. Two major types of displays were tested: three dimensional PHIGS displays and two dimensional X displays. PHIGS display performance was marginal, providing display switching speeds of just under 1 second on a VAXstation 3100 Model 76 with 32 megabytes of memory. The following tables provide timing information for sample GWA operations under a medium workload. The times were taken with a stopwatch. More comprehensive measurements will be taken in the future by timestamping the operations in the software and varying the workload. Times listed as "< 0.2" were operations that completed too quickly to measure the duration accurately. These operations appeared to provide "immediate response" from the user's perspective.

Display Switch Times

To PHIGS Display	1.0 sec.
To X Display	< 0.2 sec.

Display Update Times

To PHIGS Display	3.8 sec.
To X Display	< 0.2 sec.
Alarms	< 0.2 sec.

Perform Display Element Action

Render Pop-up Table < 0.2 sec.

The slow PHIGS display switch time appears to be attributed mainly to the large amount of processing within PHIGS itself, and is not a symptom of the generic nature of the design. PHIGS display update times are high because the current implementation allows updates no more often than every 3 seconds to avoid continuous, sequential, full display updates. Faster

hardware and better system tuning may increase PHIGS performance.

3.3 Using the GWA

A system developer uses the GWA as the basis for developing the user interface component of a command and control system. The following list summarizes the tasks a system developer would perform to integrate the GWA into a system.

- o Use the existing GWA as a rapid prototyping tool to help define display requirements.
- o Determine which system requirements are already met by the GWA and which must be developed.
- o Merge GWA documentation (currently SSSI and SRS) into the developer's documentation set.
- o Revise the GWA software platform, if necessary. We anticipate only minor modifications in the platform.
- o Revise the GWA external interfaces to match the target mission application and database system.
- o Extend the display element set to satisfy the system's display requirements.
- o Build any displays for the system that were not already built in the prototyping phase.

4.0 FUTURE ENHANCEMENTS

Initial enhancements planned for the GWA naturally include refinements to the platform architecture (incorporate unimplemented requirements such as multi-level security features), the enhancement of existing reusable software components and the addition of many other components. The goal of rapidly developing a command and control system requires a near- complete architecture and a wealthy library of components. Additional components might include various two dimensional map projections, status displays,

manual entry data forms, bar graphs, and other common display objects.

The next step in the evolution of the GWA concept is to extend the Graphic Engine and Display Builder concepts to related development areas outside of the workstation realm. This includes the message processing and data management areas that are relegated to the mission processing segment which interfaces to the GWA. We believe that the techniques applied in the analysis, design and implementation of the user interface can be applied to the message and data management design to provide a platform architecture and reusable components for the development of an entire command and control center system. The interfaces between these three major areas must be well specified and the components must be well-integrated to allow the development of tools (like Display Builder) for an effective and consistent command center design.

5.0 SUMMARY

The Generic Workstation Architecture is a system that provides a flexible user interface environment for integration into a strategic or tactical command and control system. By defining a command and control display as a composition of lower level display elements, we are able to dynamically create and modify displays by manipulating parameter files that define the displays. This approach leads to an important method for developing user interfaces. The GWA is used first as a rapid prototyping tool to define and evaluate displays. Next, the displays defined during the rapid prototyping activity are used in the operational system. Finally, display capabilities can be extended by creating and modifying displays dynamically in the operational environment.

We anticipate that significant resources can be conserved by reusing a workstation environment during C2 system development as an alternative to developing the entire user interface component anew. Further, the GWA design provides flexibility and extensibility we have not seen in other environments.

6.0 ACKNOWLEDGEMENTS

We would like to thank the entire team that helped develop the GWA concepts and current implementation. Without the hard work and dedication of these engineers, this program would not have been possible. For all of us, this was essentially a part time effort, with some periods of full time work. Thanks to Darren Stautz and Craig Baxter for their many important contributions as members of both the design team and implementation team. Thanks to Jordie Harrell for his contributions during the requirements analysis activity. Thanks to the other members of the implementation team (in alphabetical order): Charulata Kearney, Patrick Martin, Karri McCarthy, and Steve Statham. Thanks are due to Patrick also as a key member of the integration team. Thanks to George Macpherson for his thoughtful review of this paper. Thanks also to Marti Devine for her tireless support in preparing our technical documentation. Special thanks are due to Bill St. John for his overall vision and work in securing corporate support. We also thank Digital Equipment Corporation's Frank Backes and Stephen Schreifer for their assistance and contribution in establishing an adequate development environment for this project.

7.0 REFERENCES

1. Brewin, B., "CIM." Federal Computing Week (CIM) September, 1991.
2. Portable, Reusable, Integrated Software Module (PRISM) Draft Request for Proposal. Solicitation Number F19628-91-D-0016. Issued by ESD/AVK, Hanscom AFB, MA, April 30, 1991.
3. Dijkstra, E., "Structured Programming." Software Engineering Techniques, Report on a Conference Sponsored by the NATO Science Committee, p. 84, October, 1969.
4. Parnas, D. L., "On the Design and Development of Program Families." IEEE Transactions on Software Engineering, vol SE-2, no. 1, p. 1, March, 1976.

5. Brown, Gerald R., and Quanrud, Richard B., "The Generic Architecture Approach to Reuseable Software." Proceedings from the Sixth National Conference on Ada Technology, 1988.

6. Tracz, W., Contribution to "The Open Channel." Computer, Vol. 24, No. 10, October 1991.

7. Booch, Grady., Object Oriented Design with Applications. Benjamin/Cummings Publishing Co., Inc., Redwood, CA, 1991.

8. System Specification for the Mobile Command and Control System Program. August, 1989.

9. System Specification for the Granite Sentry Program, Phase II, Revision A. March 1, 1989.

10. System Specification for the Anti-Satellite Program. September 20, 1990.

11. System Specification for the Joint Space Intelligence Center Program. January 20, 1989.

12. System Specification for the Automatic Message Handling System Program. February 28, 1989.

13. Command Center Architecture (Draft). February, 1989.

The Authors:

Kirstan Vandersluis is a Senior Software Engineer at Softech, Inc. in Colorado Springs, Colorado. He received a B.S. degree in Computer and Communication Sciences at the University of Michigan, Ann Arbor, and an M.S. degree in Computer Science from the University of Colorado, Colorado Springs. His professional interests include software engineering process improvements for both Ada and C development environments, generic architectures, and software portability. He presently serves as Software Development Manager for a commercial program to develop user interface and simulation tools for a digital image processing application.

Paul Richards is a Senior Software Engineer at Softech, Inc. in Colorado Springs, Colorado. He is a lead designer and chairman of the Display Working Group for the Mobile Command and Control System / Mission Support Segment (MCCS/MSS) project for USSPACECOM. He received a B.A. degree in Computer Science from Dartmouth College and M.S. degree in Computer Science from Chapman College. His professional interests include software engineering, software reusability and graphical user interfaces.

Readers may contact either author at SofTech, Inc., 1330 Inverness Drive, Suite 315, Colorado Springs, CO 80910, phone number (719) 570-9400.

A Schema for Extensible Generic Architectures

Curtis Meadow and Larry Latour
University of Maine
Department of Computer Science
Orono, Maine, 04469

meadow@maine.maine.edu, larry@gandalf.umcs.maine.edu

Abstract

Attention has recently focused on the design and construction of generic architectures, templates that can be instantiated with contextual information to produce working systems. Such architectures are useful for constructing collections of integrated reusable components within a particular domain. Our approach allows for the systematic construction of a layered generic architecture (LGA) as well as the addition of components to the architecture. Such an architecture is thus extensible, and also adaptable, in that it can easily be modified to fit changing end-user contexts. We define a schema for such architectures, allowing us to isolate and deal with a number of orthogonal architectural issues.

1 Introduction

A good deal of work has recently been done in the design and construction of generic architectures, i.e., system or subsystem templates that can be instantiated with contextual information to produce working systems [Coh90, CAMP 85, Muss 87]. Such architectures are useful for constructing collections of integrated reusable components within a particular domain. Our approach allows for the systematic addition of components to the architecture. Such an architecture is thus extensible, and is also adaptable, in that it can easily be modified to fit changing end-user contexts. We define a schema for such architectures, allowing us to isolate and deal with a number of orthogonal architectural issues within the structured framework.

As a starting point, we draw on the 3C model of component structure [Edwa 90, Edwa 90b, FLW 90, Lato 89, Trac 90]. The 3C model is a basic reference model for the decomposition of a single component into several "subcomponents." These subcomponents

are concept, content, and context. The key to the model is the separation of contextual dependencies from both the component specification (its concept) and its implementation (content). Separation of specification and implementation is already well-known and widely practiced. However, the isolation of contextual dependencies has not been well-explored, nor has it been subjected to a formal analysis.

We contend that careful isolation of contextual dependencies in the modular structure of a generic subsystem can produce a system that is not only capable of being instantiated to a large number of actual components, but can be easily extended by "plugging in" newly constructed modules embodying different contextual information. This approach has to an extent been explored in the Common Ada Missile Packages [CAMP 85] and by David Musser and Alex Stepanov in The Ada Generic Library [Muss 87].

The modules in an extensible layered generic architecture (an LGA) are divided into several classes, based on the 3C model. Each class serves a distinct purpose within the architecture. These classes are as low-level abstractions, abstract algorithms, base abstractions, and view abstractions, described in more detail in the following sections.

2 A Methodology for LGA Design

The goal of this paper is to outline a methodology for engineering collections of reusable components. Such a methodology is not a strict set of rules, but rather a set of guidelines for design. Here we are concerned not only with the design of single components, but with the design of collections of components.

Engineering a collection for reuse implies that the collection should be more than an ad hoc grouping of components. Several domain-specific collections of components have been introduced and are

in widespread use. The X-Windows system and the Unix operating system are examples of domain-specific reusable component collections. The success of these collections of components suggests that a general methodology can be applied to any domain in order to produce a structured collection. The structure or taxonomy of a collection reflects two bodies of knowledge:

1. the knowledge base of a domain is embodied in the structure, so that a user or designer who is familiar with the domain finds a readily comprehensible structure;
2. sound software engineering practices are also embodied in the structure of a collection, independent of the knowledge base of the domain.

Engineering for reuse is the application of a specific methodology (or set of rules) to the design of a collection. The design methodology is applied to the knowledge base of the domain to produce a structured collection of components. Unix utilities exemplify this process on a small scale. A utility can be integrated into a Unix system if it follows certain rules about the treatment of input and output (reading and writing standard input and output, treating files as streams of characters, and so on) and uses established protocols for system calls.

The methodology being proposed here is not a methodology for system design, but for the design of reusable component collections. The design of a component collection may be orthogonal to the design of a system, in that one of the goals of component collections is to produce "black-box" components that can be plugged into a system. The particular structure of the collection itself is irrelevant to a system using one of the components contained in or generated by the collection.

2.1 The 3C Model of Component Structure

A design methodology provides a structured framework for thinking about design. As such, the designer must have a clear mental image of the framework; i.e., the methodology should provide the designer with a model for thinking about design. For example, the top-down design methodology for complex systems provides the designer with a model of a system as a hierarchical structure of problems and subproblems.

The layered generic architectures methodology uses the 3C model of component structure as the basic reference model of component decomposition. The

model is a view of a component as the composition of three parts: concept, content and context. The concept of a component is an abstraction of "what" the component does; the content is "how" it does it; and the context is both "where" it is done and "what it is done to."

The concept of a component is the abstract model that the component represents, i.e., the abstract specification of the component. Concept is partially embodied in an Ada module in the specification of a package; however, the concept is not the package specification. The representation of a concept in an Ada specification is incomplete—the concept properly includes the full semantic specification of an abstraction.

Concepts represent abstract machines in the same way that built-in concepts in a language are representations of abstract machines [Parn 72]. For example, the array types built into many languages provide an abstract machine, with indexing, assignment, and possibly slicing operations. The actual representation of an array is hidden from the user.

The content of a component is the algorithmic abstraction of its implementation—how the component does what it is supposed to do. Concept and context determine the content of a module. Content effectively refers only to a particular implementation, and a given concept may be implemented with many different "contents." Concept and context are a priori attributes of a module; while content is an a posteriori attribute.

The principle of separation of concept and content is already well-known, and is incorporated in the design principles of encapsulation and information hiding. By separating the content from the concept, other modules can rely on the concept (the specification) without being affected by changes in the content (the implementation). However, the separation of context from concept and content has not been well-explored.

The purpose of separating the context of a component from its other aspects is to increase the genericity of the component. The context of a component can be defined as the environment in which the component is to function, or that which completes the definition of a concept within a given environment. The context of a component is multi-dimensional, and it is necessary to classify several types of context in order to understand it fully. Three types of context can be distinguished:

1. Specification context: the set of bindings in an environment that serve to narrow a generic component to a specific component

2. **Implementation context:** the set of bindings in the body of a component that satisfy or implement a set of constraints generated by the specific environment within which the component functions.
3. **Representation context:** the set of bindings to an environment that serves as a representation for the module, or the characteristics of the representation.

A simple example of the three types of context is a generic stack package, intended for use in the presence of multiple threads of control, in which the stack is represented by a linked list. The specification context is supplied by the generic parameters that specify the type of objects stored in the stack. The representation context can be supplied by "with-ing" a list package to provide a representation for the stack, while implementation context can be supplied by a semaphore or monitor package. Using conventional programming techniques the implementation and representation context are usually hard-coded into the stack package, either by "with-ing" appropriate modules or by actually hard-coding the linked-list and semaphore abstractions into the body of the stack.

2.2 Layered Generic Architectures

The essence of our methodology is to build a collection of reusable components within a layered generic architecture (LGA), a layered collection of generic or highly parameterized modules. The structure of the collection is provided by the decomposition of "target" modules into several modules, each serving a distinct function within the architecture. In an LGA, not all modules are "created equal." For example, a queue can be decomposed into three modules

1. a low-level abstraction of a sequence (context);
2. a set of generic operations on sequences (content);
3. a set of specific queue operations implemented using the modules above (concept).

The low-level abstraction is an interface to the virtual machine provided by the programming language and is used to provide a representation for the queue. The low-level abstraction incorporates the context of the final, fully instantiated module. The generic operations in the second module incorporate the algorithmic content of the queue. The queue operations in

the third module map the algorithms for sequences to a set of algorithms for queues. Only the queue operations module is designed for the end-user; the other modules are internal parts of the architecture. In an LGA, a fourth module may be needed to bind the other three modules into a usable abstraction.

2.3 An LGA Schema

The following is a four-part schema describing the function of each layer within a layered generic architecture. It is in part derived from the Musser/Stepanov classification scheme [Muss 87].

- **Low-Level Data Abstractions and Context:** Interface to the abstract machine of the language, encapsulates implementation context
- **Abstract Algorithms:** Abstract, representation-independent algorithms
- **Base Abstractions:** Canonical abstractions of the domain; the central abstractions of the architecture
- **Views:** Mappings from base abstractions to new abstractions

Table An LGA Schema

Low-level data abstractions are the primitive data types of a layered generic architecture, just as integers, characters, floating-point numbers, and fixed-point numbers are the primitive data types of many programming languages. Low-level data abstractions capture the commonality inherent in the entire architecture. In a sense, low-level data abstractions are the lowest common denominator of an LGA. In the Ada Generic Library [Muss 87] the primitive data types are simple lists that incorporate the operations common to most sequential data structures. In the CAMP modules, the Data Types and Structures layer incorporates the operations and types needed to describe the state of every type of missile system that can be generated from the packages.

Low-level data abstractions directly provide the representation context for the LGA. Variations in implementation context can be (but need not be) implemented at this level. For example, storage management is directly implemented at this level by Musser and Stepanov, although storage management could be provided (also as a low-level abstraction) through the services of a Storage Manager module (or family of modules).

Abstract algorithms are the nucleus of an LGA. Abstract algorithms centralize the algorithmic content of the component domain in representation-independent modules. Abstract algorithms are representation-independent because they are expressed in terms of a generic type whose semantics are precisely defined at the interface by a set of operations on that type. One of the functions of a base abstraction is to map the low-level data types to the type required by abstract algorithm modules. In the Ada Generic Library, abstract algorithms are incorporated in the Linked List Algorithms package, a set of thirty-one operations on linked structures. In the CAMP modules, abstract algorithms are incorporated in the Basic Operations Layer.

The terms "base abstraction" and "view" derive from the base relations and views of a relational database. The base abstractions are a minimal set of abstractions from which other abstractions in the domain can be derived by mapping operations. Base abstractions are the canonical abstractions of the domain. In the Ada Generic Library, the Singly Linked Lists package is a base abstraction. By mapping operations, many other linked structures can be derived, such as stacks, queues, dequeues, strings and so on. These abstractions are called "views" because they map the operations of the base abstraction into a different view of the data structure.

There is in general a choice of several low-level data abstractions for any given base abstraction. A base abstraction is usually "hard-wired" to a group of abstract algorithms, since the algorithmic content of the domain tends to be invariant. Any number of views may be built on top of a base abstraction.

A generic subsystem embodies two types of content: algorithmic and structural. Algorithmic content is programming in the small; an implementation of a single algorithm or families of algorithms. Structural content is programming in the large; it refers to the scaffolding or architecture of a subsystem, and is implemented by a particular modular decomposition. Generic architectures incorporate the structural content of a domain in the structure of the architecture and isolate the algorithmic content of the domain in abstract algorithms.

3 Design Methodology

The LGA design process is primarily a process of analysis; once the architecture has been established, coding the packages is almost trivial, except for the algorithmic abstractions. The graph packages from the Booch components [Booc 87] will be used as an

example of the design process, illustrating the differences between the traditional approach (as exemplified by Booch) and the layered generic approach.

The process of constructing an LGA is basically the process of decomposing end-user modules into the 3C's of concept, content, and context. In general, the process of decomposition transforms one module into several modules, as the concept, context and content of the target module are mapped onto individual modules within the LGA.

At first glance the decomposition process would seem to produce more code and more modules rather than less code and fewer modules. However, the result of decomposing one hard-coded abstraction (for example a stack, graph, or symbol table) should be virtually identical to the result of decomposing any other hard-coded abstraction of the same class. The resulting collection of modules is capable of producing many members of the target class of abstractions. The module at the starting point of the decomposition process is a focal point for capturing the variety of design decisions that are associated with a class of components.

The following outline presents the broad steps of the design process. Each step will be discussed in detail in the sections following. The guidelines suggested here are heuristic in nature; hard and fast rules can only be stated for narrow domains.

1. Domain Analysis: determine the extent and breadth of the family of components to be generated by the LGA. This is the "top" layer of the LGA. This step determines the end-products of the LGA; the base and view abstractions, and determines the variations in specification context that might be expected.
2. Examine one or more "representative" members of the family of components, and perform a modular decomposition, using the 3C model. This decomposition should serve as a "trial" architecture, suggesting possible representation context (low-level data structures) and implementation context (constraints generated by the environment).
3. Using the results of (2), isolate and expand the algorithmic content of the LGA.
4. Specify and implement the low-level data structures.
5. Specify and implement the algorithmic abstractions.

6. Integrate the base abstractions, using the low-level abstractions and algorithmic abstractions developed in the previous steps.
7. View abstractions follow straightforwardly once the base abstractions have been fixed.

Although the methodology above is presented as a sequential set of steps, in practice the design process involves looping and recursion. The first attempts are seldom satisfactory. Note also that the methodology varies slightly when "re-engineering" existing components. The methodology presented is a top-down methodology; re-engineering existing components requires some elements of a bottom-up methodology.

3.1 Domain Analysis

The first step in designing a layered generic architecture is to examine the domain of the components to be represented by the LGA. The purpose of the domain analysis is threefold:

1. to determine the extent and breadth of the end products of the LGA;
2. to determine the variations in specification context that can be expected within the domain of applicability;
3. to determine which top-level abstractions should be implemented as base abstractions, and which should be implemented as view abstractions.

Domain analysis typically starts with the examination of a known component or abstraction. When examining a representative component, the following questions can be asked:

- What common variations of this component exist? What are the variations of concept?
- Which variations represent structurally distinct abstractions, and which are simply variations in interface specification?
- Given the results of the last two questions, what variations in the specification context can be expected? Which parameters should be left generic to be instantiated by the user, and which parameters should be partially instantiated within the architecture?
- Which abstractions are base abstractions and which are views of base abstractions? One heuristic is that in general base abstractions are "key" modules to the LGA; that is, they are the

parents of a family of variations. They represent the commonality of the family. Note that some base abstractions may be layered on other base abstractions.

3.2 Modular Decomposition

Modular decomposition should be performed on the abstractions determined by the domain analysis to be the base abstractions of the domain. The goal of the decomposition is to produce a tripartite division based on the 3C model. This decomposition should serve as a "trial" architecture, fixing variations in implementation context, suggesting possible representation context (low-level data structures) and algorithmic abstractions. After this process has been satisfactorily completed, the remaining base abstractions can be decomposed using the first attempts as a model. This stage of the decomposition is directed towards (1) actual implementation of the base abstractions, and (2) final isolation of algorithmic content in one or more packages of abstract algorithms. Some of the questions to be asked during the modular decomposition are the following:

- What are the possible variations in implementation context? Some major issues include storage management, concurrency control, import/export of objects from/to secondary storage, and efficiency considerations.
- At what level should the variations in implementation context be implemented? Some considerations associated with the major issues listed above are as follows:

Storage management variations are usually associated with the low-level data structures, because the actual representation is being managed. Concurrency control mechanisms vary. When a non-shareable component is constructed on top of a shareable component, (e.g., a stack on top of a list) concurrency control must be implemented at the level of the base or view abstraction. When the low-level components are not inherently shareable, (e.g., a table) then concurrency control can be implemented as part of low-level data structure. For persistent objects, the save/retrieve operations may need to be implemented either at the top or the bottom levels, or at both levels. Efficiency considerations in a layered generic architecture should be determined without regard to inefficiencies resulting from layering and from abstracting away context.

- What are the low-level data abstractions associated with the abstractions? The general technique is as follows:

1. Investigate commonly used representations to determine the possible choices available.
2. Investigate the effect of the representation choices on efficiency of various operations as a criterion for suitability.
3. Look for a representation which has an associated algorithmic abstraction—this will offer the greatest flexibility and the least complexity.

3.3 Testing the Modular Decomposition

Once a tentative modular decomposition has been decided upon, it should be tested for suitability. The purpose of the testing is to establish whether suitable base abstractions, low-level abstractions and abstract algorithms have been chosen. Ideally, these three abstractions will be different facets of a common concept.

Testing for a suitable decomposition can be accomplished by coding the body of a base abstraction in terms of the low-level or base abstractions and abstract algorithms, using calls to the as-yet nonexistent abstract algorithms. This process helps to isolate algorithmic abstractions. Some heuristics for the process of isolating abstract algorithms are the following:

1. All subprograms in the base abstraction should be short and concise. Long subprograms can indicate an unsuitable specification, the need for an abstract algorithm, or an unsuitable representation.
2. The representation chosen for the base abstraction will involve a set of subprogram invocations in the body of the abstraction. The set of subprograms invoked represents the algorithmic content of the domain. Examine this set of subprograms and ask the following questions:
 - Does the set of subprograms required suggest a different representation? Some heuristics are as follows: If a very small set of subprograms is required, then the representation and algorithmic content may be imported directly as a low-level abstraction directly through parameters in the specification. An incoherent set of subprograms

(no known abstraction matches the set) suggests that the representation may need revising or a new abstraction may need to be defined. If the set of subprograms matches the specification of the base abstraction, then there might be some way to generalize the specification of the base or view abstraction.

- If the set of subprograms is a subset of some known and available abstraction then is the set a subset of more than one known abstraction? Do the abstract algorithms need to be divided into several packages?

3.4 Abstract Algorithms

Using the results of the modular decomposition, isolate and expand the algorithmic content of the LGA. The purpose of the abstract algorithms packages are to isolate and concentrate the algorithmic content of the domain. Many of the algorithms that belong in these packages will be obvious; others may not be so obvious.

Something that should always be considered is the use of higher-order generic subprograms. For example, the designer of a Set package will typically include operations such as Intersection, Union, Difference, and Member. Another level of reuse can be accomplished with operations such as Select, which returns a subset satisfying a given criterion, or Detect, which returns an arbitrary element satisfying a criterion:

generic

with function Test (E : Element) return Boolean;
function Select (S : Set) return Set;

generic

with function Test (E : Element) return Boolean;
function Detect (S : Set) return Element;

3.5 Bottom-Up Design Techniques

When re-engineering existing abstractions, some additional heuristics need to be applied during the domain analysis. Part of the danger of re-engineering existing specifications is that existing specifications tend to be context-dependent. Several criteria must be applied to determine the "suitability" of a specification for a generic module:

1. Genericity

- Is the module as "generic" as possible?
- How much context is hard-coded? Can any of the context be moved out into the specification in the form of parameters?
- Will the module serve as an "end-point" of the library (a view abstraction) or will it be a base abstraction?
- Can the specification be broadened to allow variants of the abstraction to be implemented by parameterization?

2. Breadth

- Does the specification supply sufficient operations for all purposes?
- Do the semantics of the operations unnecessarily restrict the breadth of the generic module (reduce its genericity)?

3. Higher-Level Operations (Generic procedures)

- Can the specification be broadened by the inclusion of generic procedures and functions with procedure and function parameters?

4. Base and View Abstractions

- Does the module represent the parent of a family of variations? If so, it is a candidate for a base abstraction. Is there a good reason to hard-code the representation or should it be moved out into the specification as a set of parameters?

3.6 Example: Graphs

The following discussion uses the general class of graph abstractions to illustrate the beginning steps of the design process. We first examine the basic concept of the abstraction and possible variations of that concept. A graph is pair (V, E) where V is a set of vertices and E is a set of edges. A vertex is a labelled (named) object, usually (in computer applications) with some associated data (referred to below as the attributes). An edge is a pair (v_1, v_2) where v_1 and v_2 are members of V ; in other words, an edge connects a pair of vertices. Vertices are often referred to as nodes, and edges as arcs. In the following discussion these terms will be considered to be synonymous. Edges may or may not have some data associated with them (edge attributes), and they may or may not be labelled.

3.6.1 Domain Analysis

Several variations of graphs are commonly used. Graphs can be simple graphs (only one edge allowed between any two vertices) or multigraphs (multiple edges allowed between pairs of vertices; loops connecting a vertex to itself are allowed). Edges in either case can be directed or undirected. Note that it is possible in simple, directed graphs to have two edges between any pair of vertices as long as the directions differ. In any graph, edges can be weighted or unweighted. Graphs can also be finite or infinite. Infinite graphs present special problems of computability and can in general be handled only with lazy evaluation: they will be ignored in the following discussion.

Many other types of graphs have been described; however, most of these are mathematical restrictions of the basic types described above. Some of these types are bipartite graphs, connected graphs, biconnected graphs, acyclic graphs, planar graphs, Eulerian graphs, circuit graphs, and so on. These types of graphs can be implemented as views rather than base abstractions because they are subsets of the more general classes of graphs. Note that although trees are a type of graph, trees are used for an entirely different class of problems than graphs, and should not be considered at all as base abstraction in an architecture of graphs. Therefore the basic forms (variations of concept) of a graph are {Weighted - Unweighted}, {Simple - Multi}, and {Directed - Undirected}.

Note that simple graphs are a subset of multigraphs, and that a simple graph can be layered on a multigraph simply by including a test to see if an edge is already present whenever an edge is added to a graph. A generic template can use a Boolean parameter to allow both simple graphs and multigraphs to be instantiated from the same package. Note, however, that many graph problems are expressed only in terms of simple graphs, and many graph algorithms only work with simple graphs. This suggests that a separate package of simple graph algorithms will be needed.

Note that weighted graphs can be implemented as a view of unweighted graphs, by implementing the weight as an attribute of an edge. Unweighted graphs can also be implemented as a view of weighted graphs, by simply ignoring the weight associated with each edge. The first approach, implementing weighted graphs as a view of unweighted graphs, seems to be preferable. Weighted and unweighted graphs are used for different types of problems, and therefore should both be considered for base abstractions.

Note that neither directed nor undirected graphs can be easily implemented as a view of the other. The

two abstractions require fundamentally different representations in order to be reasonably efficient. However, if connectivity is the only edge attribute of interest, then undirected graphs can be implemented using a directed representation by adding pairs of directed edges for each undirected edge. This approach, although technically feasible, results in undue complexity.

The preceding discussion suggests that the directed and undirected multigraphs may be a suitable set of base abstractions for the architecture. Simple and weighted graphs are easily implemented as views of the two basic forms, although they are also canonical abstractions of the class of graphs.

The forms distinguished by Booch are somewhat different: he distinguishes Directed-Undirected, Bounded-Unbounded, and Unmanaged-Managed-Controlled forms. Boundedness and storage management are part of the implementation context, and should be supplied as contextual parameters rather than conceptual distinctions. Likewise, concurrency control and persistency (neither were considered at all by Booch) can be supplied by parameterization. Both of these will also require the layering of additional operations at the top level: concurrency control will require the usual semaphore or monitor operations and persistent graphs will require Save and Retrieve operations.

3.6.2 Modular Decomposition

Taking the undirected multigraph as a typical base abstraction, we need to examine possible variations in implementation context. Areas of possible variation are persistency, concurrency control, storage management, boundedness, and types of associated data. Each has unique implications for the final architecture.

1. Storage management for non-persistent graphs is fundamentally associated with the representation of the graph, and may also be associated with boundedness. Storage management should be implemented at the lowest level.
2. Inasmuch as graphs are shareable objects, concurrency control cannot be reliably implemented at the lowest (representation) level, but rather has to be implemented at the topmost level (view or base) level. Furthermore, correct behavior cannot be guaranteed by the component, and therefore is the responsibility of the user of the component. Therefore concurrency control can only be reasonably implemented by exporting

the appropriate interaction model and placing it under the control of the user, rather than the component.

3. Persistent graphs have two forms: a persistent form for secondary storage and a transient form for primary storage. Since the architecture will provide a wide variety of transient forms, persistent graphs can be implemented by building Save and Retrieve operations on top of a standard base abstraction, provided that the persistent graph is intended to exist wholly in one place or the other. Persistent graphs that are expected to remain in secondary storage while isolated vertices and edges are retrieved for various operations must be implemented at the lowest level of the architecture.
4. Bounded graphs can be offered as a feature of the architecture, using particularly efficient representations. Boundedness is almost always a restriction imposed by resource usage constraints rather than a feature of the abstraction itself. While there are indeed applications for graphs with a fixed or maximum number of vertices or edges, the limits on vertices and edges do not require a particular representation. Boundedness as a consequence of the implementation context must be implemented at a low level; boundedness as a variation in conceptual context can be implemented at the highest level (a view abstraction).
5. The type of data associated with a vertex or an edge is best left as specification context. However, many graph applications use labelled vertices and/or edges. Therefore the user should be able to supply both a label type and a datum type, and in the case of weighted graphs, a weight type. Note that base abstractions can export operations to set and retrieve attributes without knowing just what the attributes are. A weighted graph can be implemented on top of the base abstraction by providing operations to manipulate weights, which are then bundled with the edge attributes. Therefore the edge or vertex attributes can be partially instantiated in layers of the architecture to provide various abstractions at the top level.

3.6.3 Graph Representations

Graphs present an interesting issue of structural sharing. If an arc is represented simply as a pair of vertices, then it is possible to construct arcs that connect

two different graph objects. Booch considers this to be a violation of the graph abstraction. However, consider a graph containing two disconnected components. Each may be referred to independently (for example, by an algorithm that constructs spanning forests). The operation of adding an arc that connects the two components can be viewed from two differing perspectives. From one perspective, an arc is being added to an existing graph, connecting two different components.

From another perspective, two disconnected graphs are being connected into one. Therefore, why not allow users to connect graphs that were created independently? The problem is that many graph representations are based on a set or list of vertices. Adding an edge between nodes in different graphs may result in a structure whose contents can only be discovered by constructing a spanning forest, resulting in highly inefficient operations (such as testing vertex membership). This design decision then depends on efficiency considerations, which in turn depends on the graph representation. Some possible representations are the following:

1. Set representations. Graphs can be represented in a way that is close to their mathematical definition by a pair of sets representing the vertices and edges. The vertex set has no references to incident edges. Edges are represented as pairs of vertices, together with associated data. The efficiency of this implementation can be improved if vertex labels are required to be partially ordered (not an unreasonable constraint), so that both the vertex-set and the edge-set can be represented by ordered sets. An ordered-set representation can be used for both directed and undirected graphs, depending on whether the order of an edge's vertex-pair is assumed to be significant. Note also that it is possible to maintain edge-sets ordered both by source and destination vertex, allowing for very efficient operations that require knowledge of the source-vertices of a set of edges. This representation is equally well-suited for either simple graphs or multi-graphs.
2. Adjacency-list representations. Vertices are stored as a set. Each vertex has an associated adjacency list, listing all adjacent vertices. Edges have no explicit representation, because they can be deduced from the list of adjacent vertices. Edge attributes can be associated with the adjacency list. This approach is well suited for directed graphs, but not for undirected graphs, because an adjacency list is essentially a one-to-

many relation. It is easy to determine the destination vertex of an edge but hard to determine the source vertex.

3. Adjacency-matrix representations. Vertices again are stored as sets, and edges are represented as entries in an adjacency matrix. This approach is suitable both for directed and undirected graphs, as edge attributes can be stored in the matrix. Note that for undirected graphs, the edge attributes are associated only with the upper diagonal of the matrix. This representation is suitable for simple graphs, because it is easy to associate attributes with an edge. This representation can also be used for multiple graphs, provided that no attributes are associated with an edge. The entries in the adjacency matrix of a multigraph are integers indicating the degree of connectivity between the two vertices. Adjacency matrices offer very efficient representations for certain algorithms. Note that adjacency matrices need not necessarily be bounded in size—it is possible to implement dynamic matrices in a separate package. Furthermore, for particular applications, sparse-matrix representations may be highly efficient.

The above discussion of representations has revealed a distinction that was not apparent in the first attempt at domain analysis: the choice of representation depends partially on the attributes associated with an edge. If the edges are important only insofar as they connect vertices, then some representations (for example, adjacency matrices) are available that could not otherwise be used. Therefore the set of base abstractions should include graphs with and without edge-attributes.

Other representations are possible, but the three above are probably used more often than others. Having defined three possible low-level abstractions, the next task is to define a common low-level interface. By providing a standard interface for a low-level graph, the architecture will remain flexible for future additions.

3.6.4 Low-Level Interface

The low-level abstractions in an architecture should have identical, or nearly identical interfaces, so that they can easily be plugged into the base abstractions. Note that a graph is itself a system comprised of three abstractions: sets, vertices and edges. This suggests that the low-level components will themselves be layered systems. The discussion of graphs representations shows that vertices and edges cannot be

considered independently; however, all three of the representations require some form of a set to store vertices. Therefore the low-level abstractions of the architecture should themselves be parameterized by a set type.

A low-level abstraction needs to supply only the most primitive operations, in order to keep the interface to the base abstractions simple and to isolate the algorithmic content. The Booch interface to a graph abstraction can be used as a starting point to construct interfaces for both base abstractions and low-level abstractions. For a low-level abstraction, the following operations can be selected:

Add: Graph, Vertex \rightarrow Graph
Remove: Graph, Vertex \rightarrow Graph

Set Item: Vertex, Item \rightarrow Vertex
Create: Arc, Attribute, Vertex, Vertex, Graph \rightarrow Graph
Destroy: Arc, Graph \rightarrow Graph
Set Attribute: Arc, Attribute \rightarrow Arc

First Vertex: Graph \rightarrow Vertex
Next Vertex: Graph \rightarrow Vertex
First Arc: Vertex \rightarrow Arc
Next Arc: Vertex \rightarrow Arc
First Arc: Graph \rightarrow Arc
Next Arc: Graph \rightarrow Arc

Item Of: Vertex \rightarrow Item
Attributes Of: Arc \rightarrow Attribute
Source Of: Arc \rightarrow Vertex
Destination Of: Arc \rightarrow Vertex

One of the major problems with this interface is the design of a suitable iterator for vertices and arcs. Many graph algorithms require iteration through the vertices of a graph, the edges of a graph, or the edges of a vertex. Iterators can only be fine-tuned to a particular representation; in other words, maximally efficient representation-independent iterators for graphs are simply not possible. This is an example of a design inefficiency to be dealt with in the transition from a prototype component in a layered generic architecture to a production component in a live system.

The choice of an active iterator (First and Next) is dictated in this case by the syntactic restriction of Ada that uninstantiated generic subprograms cannot be used as actual parameters. Since the objective of low-level interface design is to arrive at a set of parameters to plug into a base abstraction, passive generic iterators cannot be used. Other languages

do not have this restriction on generic subprograms and may then allow a wider choice of iterators. The disadvantage of active iterators is that state information must be maintained in the object, adding some further complexity to the representation and complicating some of the problems of concurrency control.

Note also the design decision to provide iteration over the vertices of the graph and iteration over the edges of both the graph and individual vertices. Some graph algorithms (such as constructing a minimal spanning tree) require iteration through the edges of the entire graph while other algorithms (such as breadth-first search) require iteration through the edges of a only single vertex. While it is possible to provide only iteration over the vertices of the graph and over the edges of a vertex in the low-level abstraction, this approach can lead to highly inefficient iterations over the edges of the entire graph. This is particularly evident in the adjacency-list representation, where iteration over the edges of the graph is very efficient but iteration over the edges of a single vertex is rather inefficient.

3.6.5 Abstract Algorithms

A rich variety of algorithms are associated with graphs. Many of these algorithms are specific to one type of graph. Some examples of graph problems for which efficient algorithms are known are depth-first search, breadth-first search, the single-source shortest path problem, the all-pairs shortest path problem, computing the transitive closure of a graph, computing the strongly-connected components of a graph, testing a graph for acyclicity, topological sort of an acyclic graph, computing minimal equivalent directed graphs, finding the minimal spanning tree for a weighted graph, computing network flows, and so on. Another class of graph algorithms are heuristic algorithms that arrive at near-optimal solutions to some of the NP-complete graph problems, such as the traveling salesman problem.

In any case the point is that collections of graph algorithms can be constructed for each of the base abstractions and for some of the view abstractions in the architecture. Algorithmic abstractions are constructed in a bottom-up manner by isolating the set of operations needed to access a representation.

4 Re-Engineering for Use

The graph example shows that completely abstracting a problem from its context can lead to a number of inefficiencies, spanning the entire software life cycle

from design to implementation to runtime operation to maintenance. The methodology of engineering for reuse must necessarily encompass a methodology of "re-engineering for use."

The problem of inefficiency in reusable software is a complex problem. Several types of inefficiency can be characterized. Some types are more amenable to correction than are others.

1. **Procedure call overhead:** any procedure or function call involves a certain amount of overhead, often a significant amount compared to inline code.
2. **Parameter passing overhead:** it may be significantly more efficient to reference global objects than to pass parameters.
3. **Modular decomposition inefficiencies:** the modular design of a system may result in inefficiency because modules are "too small."
4. **Excessive generalization:** the example above suffices.
5. **Abstraction mismatches:** often it may be convenient to "violate" an abstraction for the sake of efficiency.
6. **Ad hoc inefficiencies:** not really inefficiencies, but within the context of a particular problem a much more efficient way to accomplish something exists.

5 Collapsing Layered Generic Architectures

Layered generic architectures can be used effectively as prototyping tools and can store source code and design decisions in a "normalized" form, eliminating redundant source code and centralizing changes to a system. However, the compiled products of an LGA may not be suitable for production use because of inefficiencies associated with the highly modularized design. An LGA used to prototype a system can be moved into production use by collapsing the layers into a smaller modular structure, thus eliminating inefficiencies associated with procedure call overhead and modular decomposition. The techniques used to collapse layers depend on whether modules are horizontally or vertically coupled, or coupled by parameterization. In the archetypical LGA schema base abstractions are vertically coupled to the abstract algorithms package. View abstractions are horizontally

coupled to base abstractions. Low-level abstractions are coupled by parameterization to base abstractions. Coupling by parameterization shares some characteristics of both vertical and horizontal coupling.

In general the modular structure of a system will be a directed graph. The nodes of the graph are the modules of system and the arcs represent the dependencies of the system. Whenever the in-degree of a node is equal to one, only one package depends on that node. Therefore the package represented by that node can be collapsed or merged with the coupling package—the resulting graph is homeomorphic to the graph of the original system. Extending this observation to subgraphs, whenever a subgraph is a tree the tree can be collapsed into a single node.

6 Conclusions

In general, the layered generic approach seems to be better tailored to research and development, design, and prototyping while the traditional approach seems to be better for a production shop. This is not to say that improvements could not be made on either approach.

The relationship between generic programming and the design stage of a more conventional software development approach needs to be investigated. Musser and Stepanov consider that the library of generic algorithms is in some sense a library of "executable designs." Their comments appear to be consistent with the observations above that heavily layered generic architectures may be more valuable during the design phase of project than during the production phase. Of course the design phase has a strong influence on the maintenance phase of the software life cycle. S. Edwards has observed that the benefits of software reuse only pay off well if they extend into the maintenance phase of the life cycle [Edwn 90].

References

- [Booc 87] G. Booch, *Software Components with Ada*, Benjamin/Cummings Publishing Co., Menlo Park, California, 1987.
- [CAMP 85] *Common Ada Missile Packages* (3 Volumes), McDonnell Douglas Astronautics Company, St. Louis, Missouri, 1985.
- [Coh 90] S. Cohen, "Designing for Software Reuse in Ada," *Proceedings of the Third Annual Workshop: Methods and Tools for Reuse*, Case Center Technical Report No. 9014,

Syracuse University, Syracuse, New York, 1990.

- [Edwa 90] S. Edwards. *An Approach for Developing Reusable Software Components in Ada*, IDA Tech Report P-2378, Institute for Defense Analyses, Alexandria, Virginia, 1990.
- [Edwa 90b] S. Edwards. "The 3C Model of Reusable Software Components.", *Proceedings of the Third Annual Workshop: Methods and Tools for Reuse*, Case Center Technical Report No. 9014, Syracuse University, Syracuse, New York, 1990.
- [FLW 90] B. Frakes, L. Latour, and T. Wheeler. "Descriptive and Predictive Aspects of the 3C's Model: SETA1 Working Group Summary.", *Proceedings of the Third Annual Workshop: Methods and Tools for Reuse*, Case Center Technical Report No. 9014, Syracuse University, Syracuse, New York, 1990.
- [Lato 89] L. Latour. "A Methodology for the Design of Reuse Engineered Ada Components", *First Int'l Symposium on Environments and Tools for Ada*, Redondo Beach, CA, May, 1990.
- [Muss 87] D.R. Musser and A.A. Stepanov. "Library of Generic Algorithms in Ada.", *Proceedings of the 1987 SIGAda International Conference*, Boston, December 1987.
- [Parn 72] D.L. Parnas. "Technique for Software Module Specification with Examples", *Communications of the ACM*, 15(5):330-336, May 1972.
- [Trac 90] W. Tracz. "The Three Cons of Software Reuse", *Proceedings of the Third Annual Workshop: Methods and Tools for Reuse*, Case Center Technical Report No. 9014, Syracuse University, Syracuse, New York, 1990.

7 About the Authors

Curtis Meadow received his M.S. degree in Computer Science from the University of Maine in 1990, and is currently a lecturer in the Computer Science Department at Maine. His research interests include

object oriented software development, programming languages and compilers, functional programming, and software reuse. His thesis examined aspects of layered generic architectures discussed in this paper.

Larry Latour received his Ph.D. degree in Computer Science from Stevens Institute of Technology in 1985. He is an Associate Professor of Computer Science at the University of Maine, Orono, Me. His research interests include database transaction systems, software engineering environments, and software reuse. He developed a hypertext-based system, SEER, for describing the many views of a software component (usage, specification, and implementation), and he is currently interested in implementation architectures and their relation to the output of domain analyses.

NOTES

IMPACT OF SYSTEM ADAPTATION ON GENERIC SOFTWARE ARCHITECTURES

Kathleen Gilroy
Software Compositions
3135 S. AlA, Suite 14
Melbourne Beach, FL 32951

Abstract

Some current approaches to software reuse advocate the development of standard domain-specific architectures (also called generic architectures). This paper identifies problems with current approaches involving generic architectures, and describes our research into techniques for developing more adaptable architectures. It addresses kinds of adaptation requirements, analysis and specification techniques accommodating adaptation, and initial results in identifying the impacts of change on different architectural models.

1 Problem Statement

A generic architecture is a system-level design for a family of related applications. The principal elements of a generic architecture are reusable software components (many different types of components are possible depending on the reuse method). An application belonging to the family is created by adopting the generic architecture, and then adapting the reusable components to meet

the application-specific requirements of the system.

Generic architecture approaches are promoted because development productivity can be enhanced through the reuse of such architectures. Pre-existing components are guaranteed to work within the architecture and can be reused directly, and new or modified components have a well-defined context for development [Quanrud 88]. However, in order to achieve the desired productivity improvements, the architecture which defines the component context must remain intact. Adaptations to meet specific system requirements are typically restricted in terms of modifying, extending or replacing individual reusable components.

This imposes limitations on the evolution of systems using the generic architecture (i.e., accommodating the changes which are an inevitable part of the system life cycle). When presented with new system requirements that cannot be satisfied using the existing architecture, either the requirements must be modified (or deferred) to accommodate the architecture, or the architecture must be modified until it can meet the new requirements. The first solution maximizes reuse at the cost of system capability. The second solution severely reduces the amount of software which can be reused.

This work was supported by the Center for Software Engineering (Mr. Gerald R. Brown) under the auspices of the U.S. Army Research Office Scientific Services Program administered by Battelle (Delivery Order 2614, Contract No. DAAH03-86-D-0001).

The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

Both solutions are counter to achieving overall cost-effectiveness goals. The impact is measured not only with respect to the development of new systems in the family, but also to the maintenance of existing systems employing the architecture (with maintenance consuming the major portion of software expenditures). We contend

that adaptability is more important than development productivity for providing overall cost-effectiveness over the life cycle of a system family.

2 Goals and Objectives

The ultimate goal of our research is to define techniques for developing more adaptable generic architectures. An adaptable architecture should maximize support for changing system requirements without minimizing the reuse of its constituent components. A generic architecture will be adaptable to the degree that:

- a. The differences among current and future requirements for systems in the domain are identified and well-understood
- b. Techniques for analysis, specification and V&V of the adaptation requirements associated with those differences exist and are appropriately used
- c. Mechanisms for accommodating those requirements in design and subsequent reuse of the architecture exist and are appropriately used
- d. The potential technical and cost/benefit tradeoffs are identified and well-understood

A derived objective of our research is therefore to identify the kinds of system differences which might occur. The differences among systems in a given application domain are identified through a process called *adaptation analysis*, which is part of an overall domain analysis. These differences are then expressed in terms of *adaptation requirements*, which are part of an overall domain requirements model. The domain requirements model is important because it serves as the basis for the derivation and validation of a generic architecture. Section 3 of this paper provides a classification of adaptation requirements which could be used by domain analysts to ensure that all of the kinds of future changes which the system may be required to accommodate are considered during the domain

analysis. It also discusses issues in applying the classification.

Another objective of our research is to identify system/software analysis and specification techniques appropriate for dealing with adaptation requirements. Generally, the techniques in popular use today do not directly support adaptation, such as by providing parameterization mechanisms, or mechanisms for representing selection from among alternative elements. A related problem is that multiple techniques and representations must be used to completely model a system. Section 4 of this paper discusses current techniques for domain modeling, and possible modifications to accommodate the specification of adaptation requirements.

One of our goals is to define a generalized approach to architecture adaptability, as opposed to a set of ad hoc techniques for adapting specific system instances. Also, the approach must focus on system-level architectural modeling, as opposed to implementation-level (program) architectures. However, very little supporting work has been done to date in formalizing architectural abstractions at the system level. Section 5 of this paper describes a possible classification of architectural models. It also discusses how various architectural models might be impacted by changes resulting from new system requirements.

Adaptation mechanisms are the specific techniques used to accommodate differences among instances of the system family. We are currently looking at approaches used by system/software designers to reduce the impacts of anticipated change, and strategies for designing system-level "workarounds" (mechanisms for adapting to new system requirements having minimal impact on existing subsystems). Section 6 of this paper presents some of our preliminary results.

We expect that highly adaptable architectures will be harder to develop and reuse than more application-specific architectures. A key risk of this approach is that the extra costs involved may outweigh the potential benefits. Potential cost/benefit implications of the approach are discussed in Section 7.

Table I
Kinds of adaptation requirements

SYSTEM CAPABILITIES - what the system does	
Mission adaptation	Accommodates changes in the purpose(s) of the system. Examples: handle new threats, comply with new tax laws.
Operational adaptation	Accommodates changes in the functionality and behavior of the system. Examples: automate a manually generated report, provide greater throughput.
OPERATING ENVIRONMENT - where the system does it	
Environment/site adaptation	Accommodates changes in the environment in which the system is deployed. Examples: redistribute tasks and/or information among nodes, enforce a new security policy.
User adaptation	Accommodates changes in the number and characteristics of system users. Examples: support additional kinds and abilities of users, change menu options by user role.
IMPLEMENTATION TECHNOLOGY - how the system does it	
Domain-oriented adaptation	Accommodates changes in the technologies used to implement systems in a given domain. Examples: replace modeling algorithms, convert to new database format.
Platform adaptation	Accommodates changes in the hardware and software on which the system runs. Examples: add another storage device, replace graphics software with hardware.
Methodology adaptation	Accommodates changes in the way the system is developed, maintained and/or reused. Examples: reimplement upgrades in Ada, replace design documentation standard.

3 Adaptation Requirements

Domain analysis approaches focussing only on identifying the similarities of existing systems in a domain (called a commonality analysis) [McNicholl 86], are expected to result in architectures which are more vulnerable to requirements changes. An analysis of the differences among systems in a domain (called an

adaptation analysis) is critical to developing a generic architecture which can adapt to meet the needs of future systems [Gilroy 89].

Many different kinds of adaptation must be considered by the domain analyst. We have organized them into seven major categories as indicated in Table I above. Aspects which differ among existing systems or which may change for

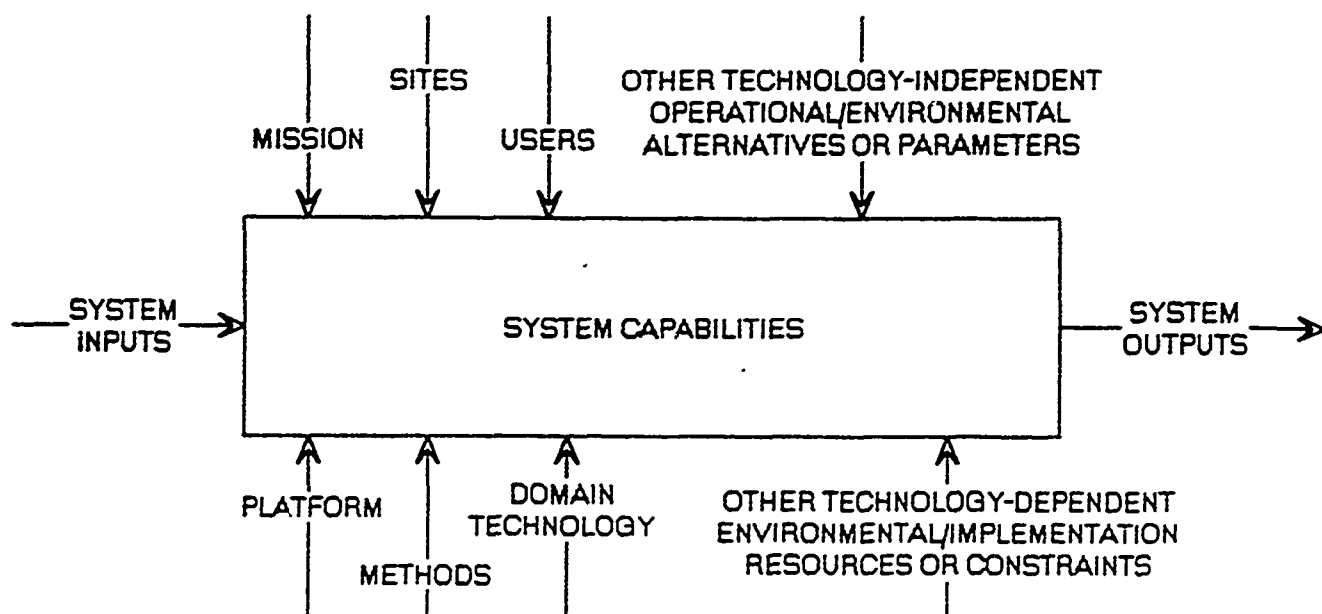


Figure 1
Another view of adaptation requirements

future systems are the source of the adaptation requirements (tradeoff analyses may later eliminate some of the potential alternatives).

We recommend that the analyses of these requirements be performed largely in the order shown (from top to bottom). This is because decisions made at a given level will constrain the alternatives available at other levels, and also limit the kinds of changes which could be supported by the resulting architecture. Specifying requirements from the lower levels early in the development process is typical (in particular, selection of hardware), and may be a primary cause of premature architecture obsolescence.

To illustrate, suppose an ASCII terminal and keyboard are selected as the user input/output devices for a system prior to an analysis of the requirements and alternatives at other levels. This (platform) decision severely limits the kinds of (operational) information which can be displayed, and the (domain-oriented) methods for user interaction which can be supported. The resulting software architecture will probably not be capable of supporting a change to a more modern

graphics workstation. Other aspects of the design were also probably influenced. For example, the information model may be oriented toward data types and groupings which can be displayed on an ASCII terminal (i.e., 24 lines of text). This even further limits the ability of the architecture to accommodate change. (Note: although the user interface and data management components are typically separated from the rest of the system in modern software architectures, this is not sufficient protection against change.)

Hatley's approach for evolving requirements to architecture recommends an ordering of decisions, but distinguishes only between technology-independent and technology-dependent factors [Hatley 87]. Figure 1 above illustrates a split of the adaptation requirements along these lines, using an SADT-like representation. The basic system consists of its inputs, capabilities, and outputs. Kinds of adaptation which are primarily technology-independent factors are shown as context or parameters to the system, and kinds of adaptation which are primarily technology-dependent are shown as resources or constraints on the system.

The relationships which are identified between the requirements at the various levels should be included in the domain model for the system family (some techniques for dealing with adaptation requirements are addressed in the next section). The importance of ordering development decisions and documenting their interrelationships is also emphasized by the feature-oriented domain analysis (FODA) approach [Kang 90].

4 Specifying Adaptation Requirements

There are at least three strategies for expressing adaptation requirements using current specification techniques:

- o Multiple specifications - this is the most typical approach, and involves creating a new specification for each different alternative being addressed
- o Annotated specifications - this approach involves annotating the elements of a specification with possible alternatives (usually with text prose)
- o Parameterized specifications - with this approach, alternative or parameterized elements are an integrated aspect of the specification

The most commonly used techniques for representing requirements are data flow diagrams (DFD), entity-relationship-attribute (ERA) diagrams, finite state machines (FSM), object-oriented analysis (OOA) models, decision tables and trees, and Petri nets. With most of these representations, only the results of a particular analysis decision can be expressed, so multiple specifications must be used to represent any differences among system instances (note: we distinguish this use of "multiple specifications" using a single representation from the need to express different views of a system using several representations).

For example, suppose a system requirement for a processing capability P specifies adaptability to accommodate either of two alternatives, A and B. If alternative A is selected, then process

bubble P in the system's DFD must be partitioned into bubbles PA1 and PA2, and if alternative B is selected, then P must be partitioned into PB1, PB2 and PB3. Two diagrams are required to represent each of the unique partitionings for a given alternative. Another example is if there were differences in the relationships and attributes of an entity E depending on which alternative was selected. In addition to two ERA diagrams showing the specific attributes and relationships associated with each alternative, a third diagram showing all of the possible attributes and relationships of E would be useful. Separate documentation would be needed to express information related to the adaptation requirement, such as to bind each diagram to its associated alternative, or to describe constraints on other portions of the system that must interface with P or E.

A specification strategy involving annotations could be used to document information related to adaptation requirements that cannot be expressed graphically. Annotations could range from English prose (such as to document a description of the alternative partitionings for process P) to formal languages (such as to document assertions governing the existence of the attributes and relationships of entity E). In some cases, annotations could eliminate the need for multiple diagrams. For example, annotations on process P would not eliminate the need for multiple lower-level DFDs, but existence constraints on the attributes of entity E would eliminate the need for multiple ERA diagrams.

Parameterized specifications generally refer to any representation that directly supports some mechanism for expressing system alternatives. A powerful technique for distinguishing and organizing system alternatives is inheritance. The differences between generalized and specialized elements, and between alternate specialized elements can be used to define required adaptations for the system. The modeling technique must also support the concept of aggregation, such that alternative compositions of elements can be expressed.

One graphical representation which directly supports the specification of options and alternatives is the "feature diagram," developed

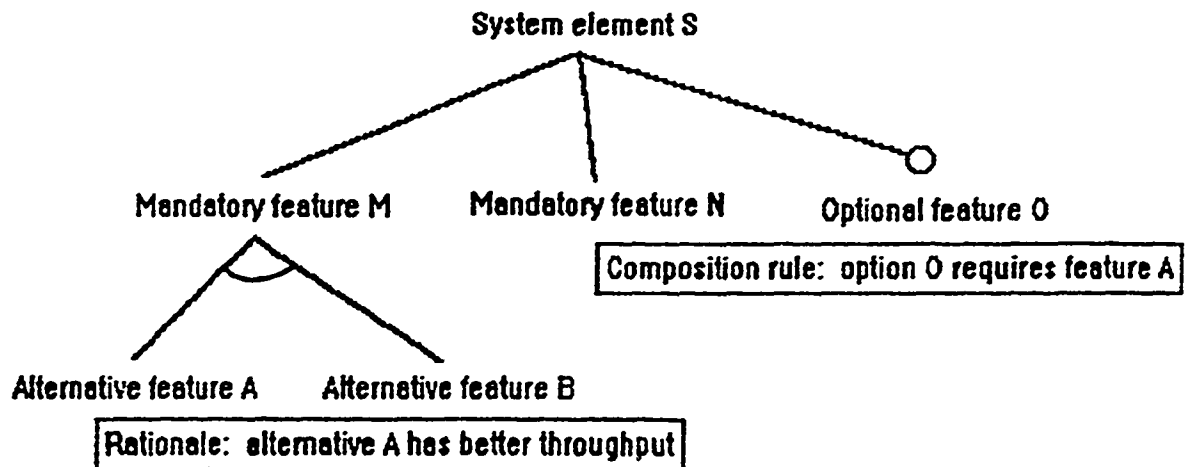


Figure 2
Example FODA feature diagram

for use with the FODA method [Kang 90]. A feature diagram illustrates the mandatory, alternative and optional characteristics of a system as viewed by the end user. The selection or combination of features may be further constrained using text annotations defining the composition rules. Annotations are also used to provide rationale used in decision-making. Figure 2 illustrates the constructs provided by a feature diagram.

Currently, FODA feature diagrams can only express a limited number of relationships, lack mechanisms for managing diagram complexity, and have incomplete automated support for the model's semantics. The results of the FODA work also indicate that increasing the adaptability of systems will significantly increase the complexity of associated requirements analysis processes and products. Analysis and specification of even the smallest systems will require the support of sophisticated automated tools. Despite these problems, we recommend feature analysis in addition to the more traditional modeling approaches. In particular, its support for composition provides a unique mechanism for expressing potential system adaptations.

Several domain analysis approaches recommend the use of object-oriented or entity-relationship modeling methods to derive a domain model. The

FODA method recommends Chen's entity-relationship modeling method augmented with generalization and aggregation concepts. [CTA 88] also recommends ERA modeling, but with annotations on entities which describe the external and internal functions provided by the entity, making this model closer to an OOA model. [Gilroy 89] describes the use of object-oriented modeling method which incorporates a rich set of semantic relationships. However, the relationships are not explicitly discussed as a method for documenting potential adaptations of a system. Instead, the adaptation requirements are documented as textual annotations on each object in the model. We recommend an object-oriented modeling method such as defined in [Gilroy 89], but with explicit consideration of interobject relationships as a mechanism for illustrating system differences.

Functional and behavioral aspects of a domain model are often described using data flow and state transition modeling approaches. Under the FODA method, these aspects are parameterized by feature for different systems, and also by issues/decisions associated with the selection of particular domain technologies. The specific representations used to accomplish this are Stateate activity and state charts, with system differences parameterized using Stateate conditions. One problem with the Stateate

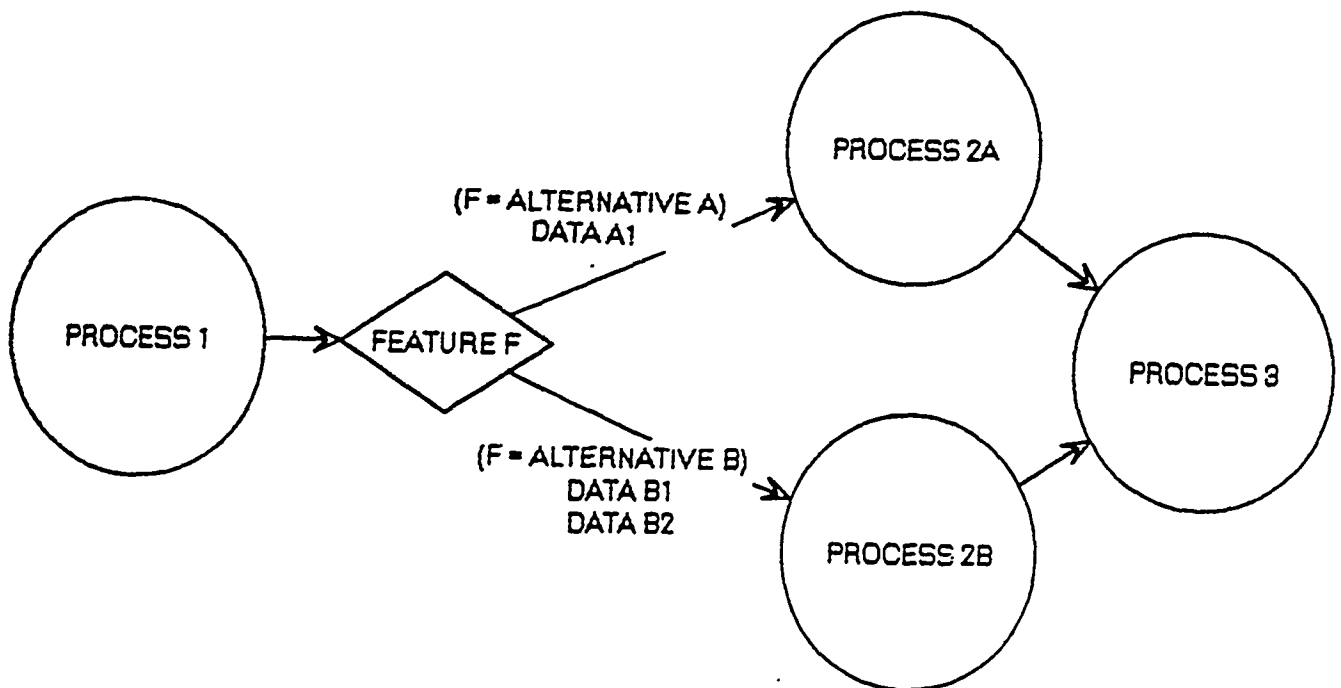


Figure 3
Parameterization of system functional specification

approach is an inability to distinguish conditions associated with adaptation parameters from other conditions relevant to the system operation. Also, generalization relationships cannot be represented graphically with this model, so textual annotations must be used.

The object-oriented modeling approach defined by [Gilroy 89] addresses behavior in terms of stimuli/responses and data flows, but does not address the parameterization of those flows. We recommend use of an object-oriented model for expressing system functionality and behavior for consistency with other views of the system, but augmented with a notation for expressing potential functional adaptations. A technique similar to that illustrated by the parameterized DFD of Figure 3 could be applied.

Performance characteristics (e.g., timing and sizing) and resource management are key considerations for real-time applications, but most specification approaches are very weak in their treatment of such requirements. Identification of techniques for specifying adaptation requirements dealing with such

characteristics, and for evaluating the impact of other adaptation requirements on system performance will require a significant amount of further research, and are deferred for now.

5 Architectural Models

A concept critical to this research is that of an "architecture." An architecture, as used in this paper, refers to a representation of the design of the system software. It defines the individual software elements of a system, their relationships to each other, and their relationships to external elements comprising the rest of the system.

The development of an architecture, as used in this paper, begins during systems analysis, when the allocation of requirements to hardware vs. software vs. manual is made. The highest level software architecture results from this allocation. The representation of a high level software architecture illustrates the major functional, performance, information and interface elements to be provided. The lowest level

software architecture identifies the specific program elements which will implement the system. Such an architecture is typically the result of detailed software design. The focus of this research is on software architectures at higher levels of abstraction (i.e., system-level architectures rather than program-level architectures).

It is recognized that there are no clear boundaries between system design and software design (for example, with "software first" approaches), between programming and software design (for example, with "Ada-based" design approaches), and between requirements and design (practitioners frequently cite the difficulty of avoiding designing while specifying requirements). However, assuming arbitrary boundaries can be applied, this research is considering the evolution of system architecture to program architecture, mappings of requirements to design to program (in particular, adaptation requirements), and the impacts of change to all three.

As noted earlier, a generalized approach to adaptable architectures requires an understanding of the kinds of architectural elements used and ways in which they may be composed into subsystems and systems. Unfortunately, a commonly recognized set of architectural abstractions has yet to be defined.

[Shaw 90a] was used as a starting point in defining a baseline set of abstractions with which to proceed. Our comparison of Shaw's taxonomy with a variety of published architectural descriptions and modeling methods prompted some modifications and the addition of new architectural categories. Table II describes the resulting classification of architectural models, and Figures 9 to 15 illustrate some examples of each model (see following pages). We have also derived a classification of architectural component kinds and composition mechanisms synthesized from [Shaw 90b] and [Merlet 90], but these will not be described in this paper.

We further abstracted the architectural models to yield the structural patterns shown in Figures 4 through 8 (shown on this page). We believe all software system architectures probably



Figure 4
Monolith abstraction



Figure 5
Chain abstraction

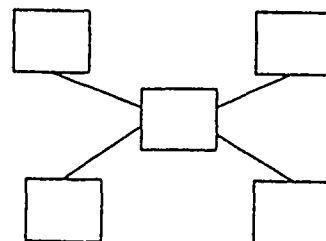


Figure 6
Star abstraction

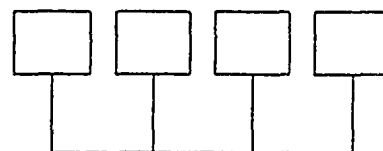


Figure 7
Bus abstraction

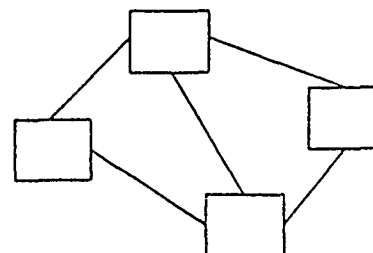


Figure 8
Net abstraction

Table II
Kinds of architectural models

Object-oriented	Independent components provide operations and maintain state. Components communicate via message passing. A major variant of this model incorporates inheritance. Figure 9 illustrates an example of this model.
Transform-centered	Independent components process an input stream and produce an output stream. Components are serially executed. Figure 10 illustrates an example of this model, commonly called a pipe.
Layered	Hierarchically organized components provide services to higher layers and use the services of lower layers. Figure 11 illustrates an example of this model.
Sliced	Distributed components provide specialized processing. Components communicate via a common bus. Figure 12 illustrates an example of this model.
Transaction-centered	Independent components operate on shared data. Execution of components is based on an input transaction stream. Figure 13 illustrates an example of this model.
State-based	Independent components operate on shared data. Implicit execution of components is based on the current state of data. Figure 14 illustrates an example of this model, also called a blackboard.
Interpreted	Interpreter processes inputs and produces outputs based on rules/facts. Figure 15 illustrates an example of this model. Not sure this is an "architecture."
Hybrids and combinations	Combinations of any of the above. For example: layered blackboards, object-oriented slices, and piped layers.

predominantly exhibit one of these structural patterns. The chain architecture corresponds to the basic transform-centered and layered architectures, with data streams being the composition mechanism in the former case, and service calls being the composition mechanism in the latter case. The star architecture encompasses the basic transaction-centered and state-based architectures. Transaction-centered architectures can be classified as a star from two perspectives: process control (the transaction center is a master governing execution of slave

operations) and data access (the database is a centralized repository accessed by all the operations). In their purest form, components of state-based architectures only communicate via the shared data. Often, an execution engine is also included with the data manager, which handles execution sequencing of the components based on the state of the system.

Object-oriented architectures are classic examples of arbitrary networks. Sliced architectures epitomize the bus abstraction.

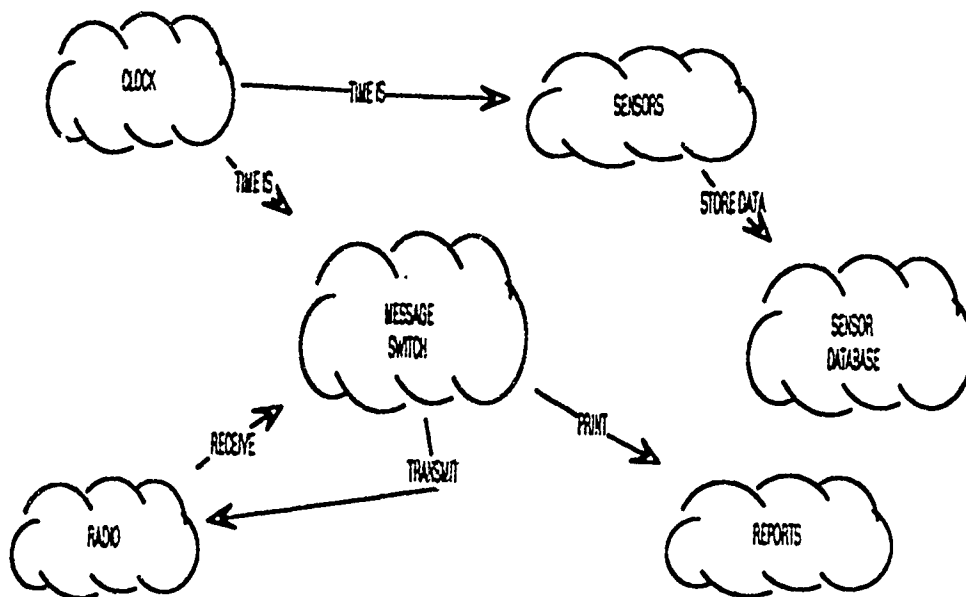


Figure 9
Example object-oriented architecture



Figure 10
Example transform-centered architecture

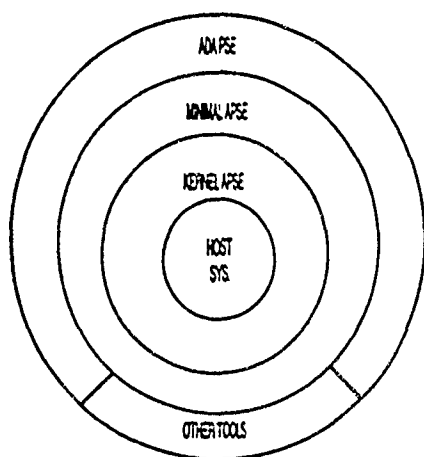


Figure 11
Example layered architecture

Interpreted systems are allocated to the monolith category because no separate application-oriented components can be readily identified (the application "components" consist of an unordered set of relatively small-grained rules/facts). A similar issue is raised for applications consisting of a formal grammar. In all the other models, components represented relatively large-grained elements of the application system (application-independent components like "execution engines" are usually only implied).

Further analysis and refinement of this model is on-going. Issues currently being addressed include: control-oriented vs. data-oriented components and composition mechanisms, implicit

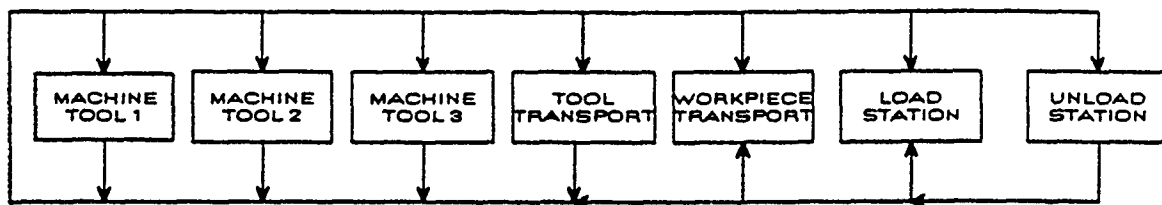


Figure 12
Example sliced architecture

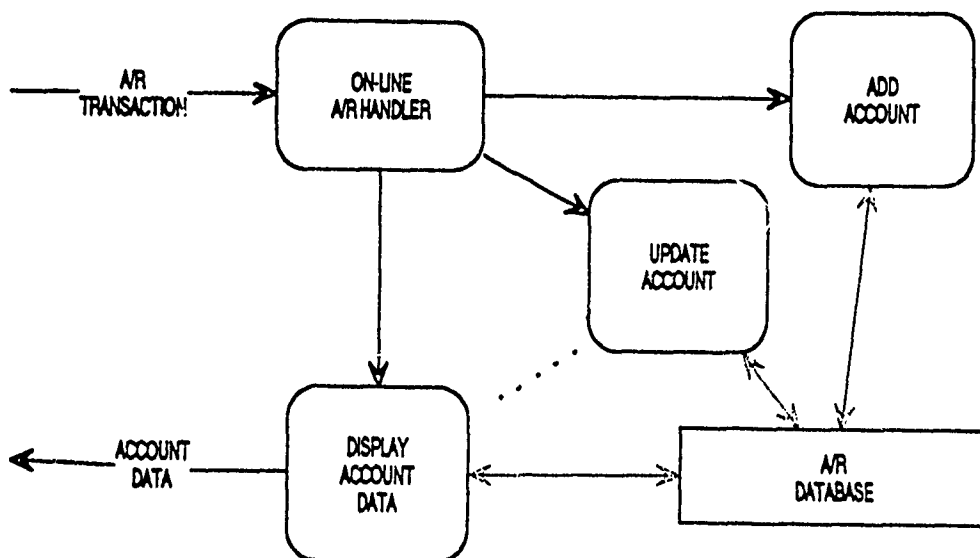


Figure 13
Example transaction-centered architecture

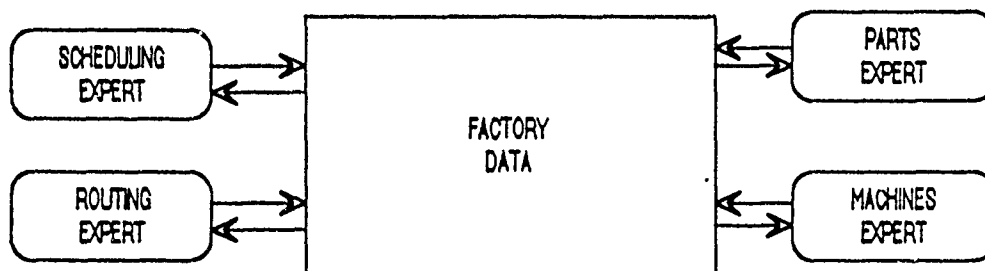


Figure 14
Example state-based architecture

vs. explicit components and composition mechanisms (for example, piped architectures merely imply a controller to accomplish the serial execution of components), and generalization and aggregation relationships between components and composition

mechanisms (for example, object-oriented models employing inheritance).

One interesting issue is that architectures can sometimes be reclassified under different

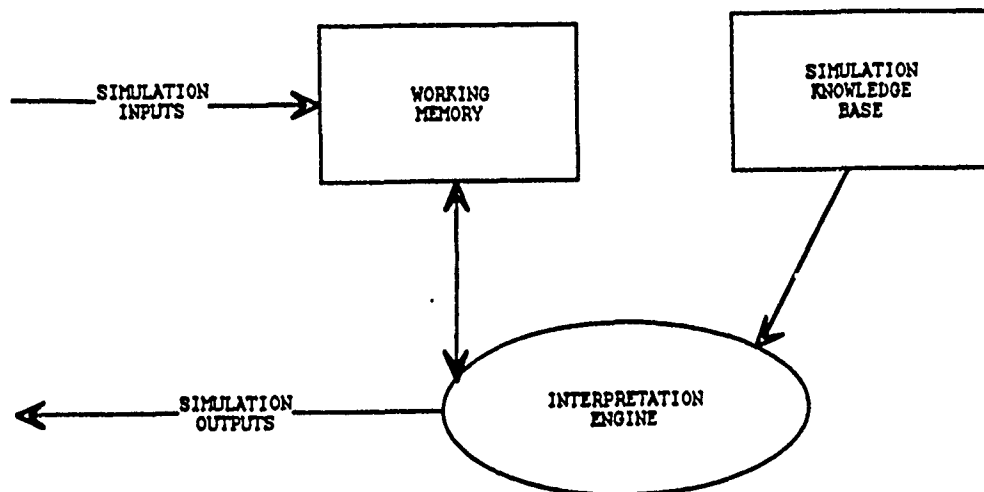


Figure 15
Example interpreted architecture

models by regrouping lower-level elements into different higher-level elements. This usually happens when critical aspects of the architecture are implicitly defined (e.g., employs an unspecified control mechanism), or components are poorly defined (e.g., has components or uses composition mechanisms that exhibit characteristics of more than one kind). It might also be an indication that the models are incorrect, or that there is a potential for merging the models into a smaller core set of architectures. However, no conclusion can be drawn simply from the identification of structural patterns when there are so many other characteristics of software architectures which must be considered.

Another aspect of this research is to investigate the kinds of changes that can be accommodated by the different architectural models. For example, consider the pipe architecture. A requirement for change which impacts the data streams processed by the system will minimally impact those components which have that stream as an input or output. Additional components might be impacted if there are derived requirements which affect other data streams in the system. Alternatively, a new filter could be developed to convert the new data stream to the formats expected by components already existing in the system. Yet another possibility is that a second filter could be added to handle input or output of just the new aspects of the data stream

(providing the changes are additive). A requirement for change in the processing performed by a component might be handled by modifying or replacing just that component (provided the content and semantics of the data streams are not affected). A requirement to increase system throughput could affect all the processing components in the pipe. However, the analyses are simplified by the knowledge that the components execute sequentially.

As another example, consider the layered architecture model. In evolving such an architecture, it is important that the chain structure be maintained. Suppose layer A uses the services of layer B, which uses the services of layer C. And suppose a change requirement is introduced for layer A which requires some service which is provided by layer C, but not by layer B. The "quick-and-dirty" approach would be to modify layer A to directly access the services of C. However, implement several such quick-and-dirty changes, and the original architecture is soon destroyed. The "correct" approach would be to modify layer B to "pass through" the required services from layer C, or possibly to "move" the service from layer C to layer B.

Other examples of adaptation considerations influencing architectural design are that object-oriented architectures are more amenable to changes in data structures (since they are hidden from users of the object), but can result in the

Table III
Initial classification of adaptation mechanisms

Parameters	Adapted by passing parameters internally.
Data-driven	Adapted from an external data source.
Language-driven	Adapted using a table or grammar (requires an interpreter).
Templates	Adapted by replacing general-purpose elements with application-specific elements.
Variant selection	Adapted by selecting from a predefined set of alternatives.
Specialization	Adapted by adding capability to a more general version. (Note: generalization is the opposite, abstracting away from a more specific version.)
Generation	An adapted version is created by a tool or well-defined manual process (could include support for any of the above).
Custom rewrite	An adapted version is manually created.

proliferation of components to manage similar but not equivalent data structures [Oskarsson 89]. Models supporting processing components as parameters are more amenable to changes involving environments and services than models which do not (also [Oskarsson 89]). However, if the selected implementation technology cannot support processing components as parameters (e.g., Ada), then the architecture may be difficult to implement or modify.

It is important to note that for the examples given in the paragraphs above, the requirement for a system change has already been analyzed to the point where the specific kinds of architectural elements which are affected have been identified (e.g., the need for layer A to access the services of layer C may have been derived from an original change requirement like "add CPU usage information to report X"). It would be impossible to identify, much less analyze, all the possible permutations of change combinations which might be encountered by system change requests. However, it is hoped that our work on classifying

adaptation requirements, architectural models, and adaptation mechanisms will provide a general-purpose framework within which system requirements changes can be planned for, analyzed and implemented.

6 Adaptation Mechanisms

Adaptation mechanisms are the specific techniques used to accommodate differences among each system instance. A classification of adaptation mechanisms is described in Table III above. This classification was based on an earlier analysis of mechanisms in use for adapting Ada programs [Gilroy 89]. One obvious issue is whether the mechanisms are appropriate for or can "scale up" to system-level architectures. Our initial analysis indicates that they are appropriate and scalable to the kinds of architecture models described earlier.

However, the research also yielded folklore on strategies for implementing system-level

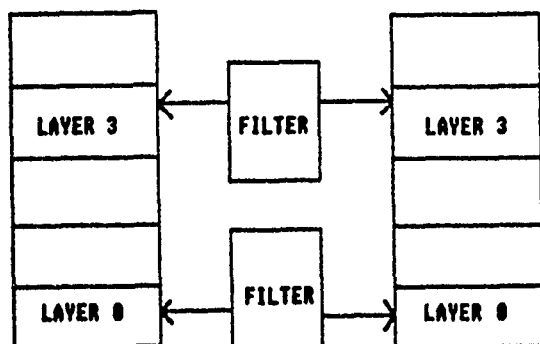


Figure 16
Filter workaround

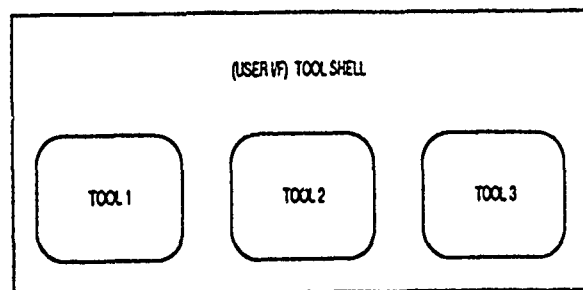


Figure 17
Shell workaround

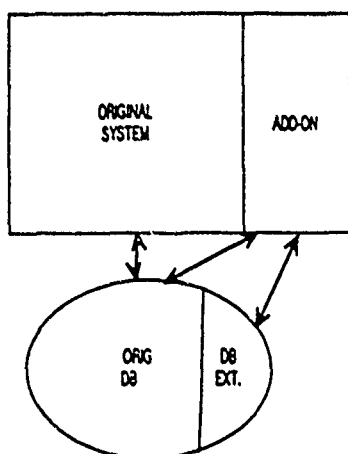


Figure 18
Add-on extension workaround

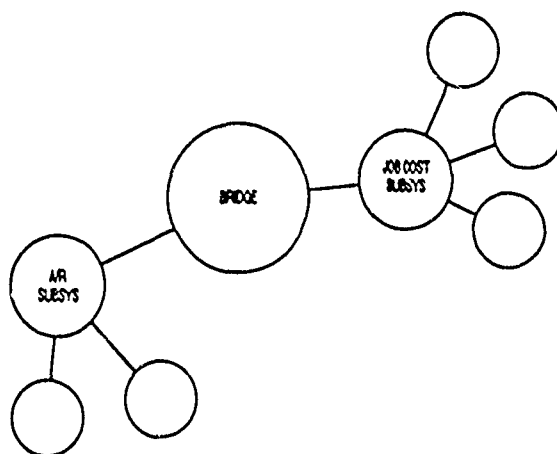


Figure 19
Bridge workaround

"workarounds" (mechanisms for adapting to new system requirements with minimal impact on existing subsystems) which do NOT seem to fall within these categories. Four examples illustrating some of these strategies are shown in Figures 16 to 19 above.

Just as the timing of decision-making in selecting adaptation requirements is important (see section 3), so also is the timing of decision-making in selecting adaptation mechanisms. A key issue in component composition is "binding" time. Table IV (next page) provides a classification of binding times. The classifications are also applicable to Ada program architectures, basically corresponding to compile-time, link-time, elaboration-time and run-time bindings of Ada components.

Binding time decisions can directly impact the software architecture and vice versa. For example, if a set of configuration parameters needs to be established on each execution of the system, then data-driven adaptation mechanisms like data files or interactive queries must be incorporated into the architecture. If the configuration parameters need only be defined once per site and never changed, then static mechanisms like variant selection can also be considered (and might be selected over more adaptable alternatives for performance reasons). A software architecture which hides design decisions is also likely to defer decisions relating to binding time. Deferring binding time provides much greater flexibility to the developer in selecting an appropriate approach, and to the maintainer who may have to replace that aspect of the design.

Table IV
Classification of component binding times

Static/fixed	A particular set of design elements is selected at the time of system creation.
Static/changeable	Alternative configurations of design elements may be selected when a particular system build is formed.
Dynamic/fixed	Design elements are bound during system initialization. Once bound, they are not changed.
Dynamic/changeable	Design elements are bound at run-time. Elements may be added, deleted or replaced during execution of the system.

7 Cost/Benefit Implications

Major cost/benefit categories to be considered by future research tasks are costs (or benefits) associated with:

- o Developing the architecture
- o Adapting and reusing the architecture
- o Maintaining and evolving the architecture

A key hypothesis of this research is that adaptability is more important than development productivity in providing cost-effectiveness over the life cycle of systems in the domain. This is based on the assumption that development productivity is higher when architectures are more application-specific, and maintenance and reuse productivity are higher when architectures are more adaptable.

However, there is a risk with adaptable architectures that less savings will result from use of this approach over more application-specific architectures, since they are harder to develop and can be harder to reuse. They can be harder to reuse, for example, when the number and types of parameters is very large. This can happen as the architecture is made more parameterized to accommodate a greater number of application systems, as illustrated in Figure 20.

However, the potential need to change the architecture to accommodate a particular system's requirements is reduced if more applications can be addressed by the architecture as illustrated in Figure 21. Our research must therefore address the concept of an ideal level of parameterization of a generic architecture to achieve the expected cost savings.

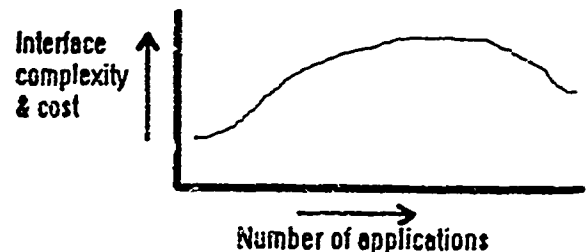


Figure 20
Interface complexity vs. number of applications

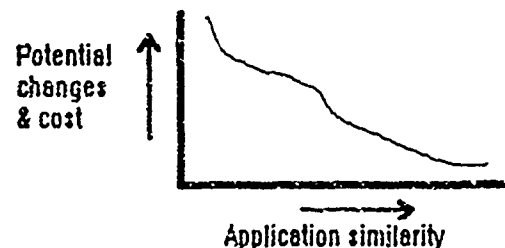


Figure 21
Potential changes vs. application similarity

8 Summary

Generic architecture approaches are promoted because they are believed to provide cost-effectiveness over the life of systems developed using the architecture. However, current approaches to generic architectures tend to emphasize development productivity and fail to consider evolution of the architecture during maintenance. It is the author's contention that adaptability of generic architectures is a more important characteristic in terms of providing life cycle cost-effectiveness. An approach to developing more adaptable generic architectures must address:

- o Identification of differences among systems in the domain
- o Techniques for analysis, specification and V&V of adaptation requirements associated with those differences
- o Mechanisms for accommodating those requirements in the design and subsequent reuse of an architecture
- o Associated technical and cost/benefit tradeoffs

Work done to date to support the identification of system differences includes a classification of adaptation requirements which could be used to guide a domain analysis. It was determined that the various classes of adaptation requirements should ideally be addressed in a particular order. Additional research is needed to further refine the classification and ordering considerations.

An analysis of popular requirements specification approaches found that they generally fail to support adaptation. Some strategies for adapting the techniques to address adaptation parameters and alternatives were identified. More work remains to be done to develop recommendations with regard to selection and enhancement of specific techniques.

Before specific architectural alternatives or other adaptations of architectures can be

determined, a more formalized understanding of possible architectural models, their components, component composition mechanisms, and architecture adaptation mechanisms must be developed. Very little external work has been done in these areas. Initial classifications in each of these areas has been developed, but the results are very preliminary and much more work is needed. A critical issue is whether "program architecture" models and mechanisms can scale up for treatment as "system software architecture" models and mechanisms. Initial results indicate that there are similarities, but also significant differences.

We have not yet done much research in the area of cost/benefit tradeoffs associated with the adaptable architecture approach. This paper described a few of the considerations, but much more work remains to be done.

References

- [CTA 88] Anonymous. Domain Analysis for Control Center Software, Computer Technology Associates, September 1988.
- [Gilroy 89] Gilroy, K., et. al. Impact of Domain Analysis on Reuse Methods, Software Productivity Solutions, CECOM CIN C04-087LD-001-00, November 1989.
- [Hatley 87] Hatley, D. and Pirbhai, I. Strategies for Real-Time System Specification, Dorset House Publishing, 1987.
- [Kang 90] Kang, K., et. al. Feature-Oriented Domain Analysis (FODA) Feasibility Study, Software Engineering Institute, CMU/SEI-90-TR-21, November 1990.
- [McNicholl 86] McNicholl, D., et. al. Common Ada Missile Packages (CAMP) Volume I: Overview and Commonality Study Results, McDonnell Douglas Astronautics Co., Technical report AFATL-TR-85-93, May 1986.
- [Merlet 90] Merlet, P., et. al. Reuse Tools to Support Ada Instantiation Construction, Software Productivity Solutions, CECOM CIN: C02087KV00100, June 1990.

[Oskarsson 89] Oskarsson, O. "Reusability of Modules with Strictly Local Data and Devices — A Case Study," Software Reusability, Volume II: Applications and Experience, Biggerstaff and Perlis (eds.), ACM Press, 1989.

[Quanrud 88] Quanrud, R. Generic Architecture Study, SofTech, Inc., Report 3451-4-14/2, January 1988.

[Shaw 90a] Shaw, M. "Larger Scale Systems Require Higher-Level Abstractions," Proceedings of Fifth International Workshop on Software Specification and Design, IEEE Computer Society, 1989.

[Shaw 90b] Shaw, M. "Elements of a Design Language for Software Architecture," Carnegie Mellon University, May 1990.

About the Author

Kathleen Gilroy is the president of Software Compositions, a small company in Melbourne Beach, Florida focussing on Ada and software reuse. In addition to her work on adaptable architectures, Ms. Gilroy is currently investigating the application of preventive maintenance tools and techniques to Ada software reuse. Over the last 10 years, she has participated in many Ada application and tool development projects, and has been a member of several working groups dedicated to evolving Ada and reuse technology.

Ada APPLICATIONS IN AERONAUTICS & SPACE SYSTEMS DEVELOPMENT AT NASA PANEL

Moderator: Carrington H. Stewart, NASA JSC

**Panelists: Anastacio M. Baez, NASA Lewis
Royal G. Bivins, NASA Headquarters
Stephen Gorman, NASA JSC
Robert Kudlinski, NASA Langley
Gary K. Raines, NASA JSC
Robert D. Steele, JPL**

DYNAMIC CONFIGURATION WITH Ada

R. Gerlich

Space Electronics Division
Dornier GmbH P.O. Box 1420
D-7990 Friedrichshafen Germany

Abstract: During its operational phase software is subject of changes due to

- **maintenance**
in order to adapt a system to new or changed requirements, to change memory structure due to hardware errors, or to improve existing software,
- **software reconfiguration**
either - on one processor - by changing the set of executable procedures, programs and accessible data or by migrating programs between several processors.

For time-critical systems on-line changes of software are required, i.e. the capability for Dynamic Configuration supporting:

1. update of a running program from version N to N+1, and
2. fast and synchronised transition from Mode A to Mode B in a continuously running task by logical instantiation of mode-specific task-bodies.

However, Ada does not support Dynamic Configuration. Therefore use of Ada is excluded in such application areas.

Dornier's solution [©] - solving this problem - consists of

- a **configuration strategy** for procedures / functions and data allowing on-line changes in real-time,
- a **software engineering strategy** guiding an engineer how to produce Ada source code which is ready for Dynamic Configuration,
- a **toolset** supporting implementation of the strategy.

It is

- designed for use in a **preemptive hard real-time** environment,
- it is **compliant** with Ada's checking and verification capabilities, and
- is supporting **existing** Ada source code.

Transition from a software configuration to another one takes a few microseconds (about 3 µs for an Intel 80386, 20 MHz). The overhead per procedure call / data access amounts 2 .. 3 µs.

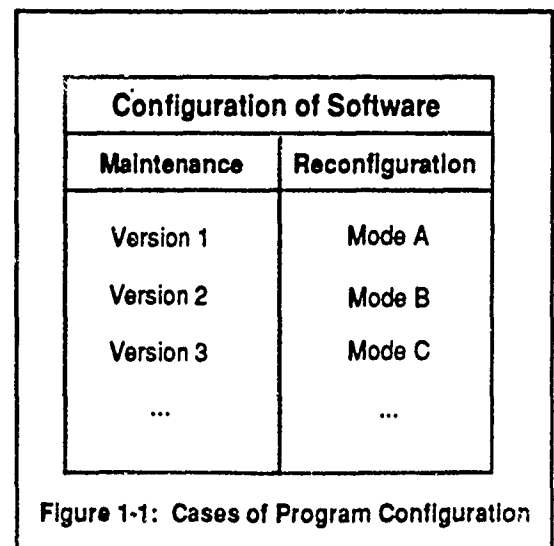
Index Terms: Dynamic Configuration, Ada, Real-Time Processing, Run-Time-Linking, On-Line Software Updates, Mode Transition, Embedded Systems

1. Introduction

The intention of this paper is to describe the technical capabilities of the concept for Dynamic Configuration with Ada and to demonstrate its feasibility. For the solution only useful features from object-oriented Dynamic Binding are introduced. It is not intended to make Ada a fully dynamical language. Only those features are provided which are needed to reduce complexity of technical real-time systems by Dynamic Configuration, and which are compliant with Ada's philosophy.

1.1. Issues for Dynamic Configuration

In technical systems, especially in embedded systems, software is the "heart" of the system. If software is stopped, the system is "dead", control over the system is lost. In case of time-critical applications, e.g. telecommunication networks, air- and spacecraft control, nuclear power plants, air traffic control, ..., the gap in control must not exceed a very limited time.



For configuration purposes (Fig. 1-1) usually software must be shut down due to the need of program

reloading. However, this takes an unacceptable amount of time. In case of Dynamic Configuration, software can continuously execute, no real gap in control appears.

1.2. Principle Concept

Dorner's concept¹ allows complete verification of the updated software before it is included into the running program. It ensures that a program, which is updated on-line, remains correct, even in a preemptive real-time environment.

The mechanism used for on-line configuration provides the capability to switch between different logical programs instantiated out of one physical Ada program. Out of a set of procedures / functions and data the logical program can be instantiated using Run-Time-Linking. On-line switch to another logical program just means to change the links between procedures and data. In case of a switch, the executing program is removed and the next one is instantiated, but on a logical level only. Therefore no time delay occurs. The physical exchange of software is mapped onto a logical level.

System dynamics is decreased because system resources are allocated by the overall Ada program only at program's start. They have not to be allocated during operation like in case of load and exchange of different physical programs². Furthermore, the same (overall) program can be used as task body for all tasks. Usually, for each combination of modes and tasks an own body has to be provided and it has to be loaded / activated and deactivated / removed each time code update or mode switch occurs.

Basically, the strategy and concept of "Dynamic Configuration with Ada" intended to support time-critical on-line applications. But it turned out, that it will simplify from an operational point of view non-time-critical applications as well, which need mode transitions.

1.3. Intended Conceptual Limitations

In order to increase reliability, it is not the intention to provide all the capabilities usually understood by Run-Time-Linking or Dynamic Linking in sense of Object-Oriented Paradigm. To do this may really be dangerous in view of reliability. Therefore only management of dynamic links is supported by the concept. This includes protection against inadvertent user access and prevents use of non-validated addresses.

¹called DC+Ada ©

All rights reserved by Dorner GmbH, 1990, for application of the concept with Ada or other programming languages

²Dynamic resource allocation by a program at run-time - foreseen by the programmer - should be avoided, in general, as far as possible from a software engineering point of view.

2. Problem Analysis

2.1. Reliability Considerations

In case of a "conventional" program the links are determined at compilation- and link-time and they remain fixed at run-time. The only possibility to update or reconfigure existing software is to load a new program under control of an operating system.

Fixed links, which have been verified by the compiler and linker, are a sufficient condition to ensure that only validated addresses are used at run-time. But it is not the only condition guaranteeing a program's integrity. In case of dynamic linking integrity can also be guaranteed provided that

- all addresses used for procedures, functions or data access are completely validated when used and
- management of these addresses is sufficiently protected against inadvertent user access.

In this context we have to keep in mind that compilers may have faults as well. But we consider them as sufficiently reliable, because compilers have been extensively tested, especially in case of Ada by well-known test procedures.

As the algorithms needed to implement Dynamic Configuration are simple, validation of related software can also be done with sufficient reliability.

Furthermore, we have to take into account that the reliability of a program does not only depend on the reliability of a compiler. It depends - at most - on the reliability of the application program. This is the most unreliable part.

And we know, that the less complex algorithms the higher is reliability of their implementation in software. By Dynamic Configuration the complexity of operational procedures to update and reconfigure a system become less complex due to the continuously running program:

- no status data are lost
- no checkpointing of data are needed
- no start / stop of a program or task is needed for reconfiguration
- total number of tasks and task bodies is reduced
- coordination of activities during a mode transition becomes simpler.

Therefore the total reliability (and availability) increases if Dynamic Configuration is used.

2.2. Update and Reconfiguration of Software

Update of software means to remove and/or to add software for maintenance reasons, i.e. the memory structure of a program is changed.

Reconfiguration of software means activation /

deactivation of software units (procedures, functions, data) within a program, i.e. changing links to procedures and data, but without structural changes of memory structure.

Therefore software reconfiguration is simpler to support than software update. In case of software update we have to ensure in addition, that fixed links, i.e. links which are not dynamically established, are not affected by structural changes in memory.

3. The Software Engineering Strategy

3.1. Principles of the Strategy

Two principle decisions were made: For provision of Dynamic Configuration

1. no compiler/APSE modification should be needed and
2. existing Ada source code should not be excluded.

These two demands directly lead to pre-processing of Ada source code prior to compilation. Of course, that does not exclude that the pre-processor can be integrated into an APSE. If desired, it is very easy.

Support packages provide the functionality needed for Run-Time-Linking, export/import of updated code and address verification.

In Fig. 3-1 the principle steps of the strategy are shown.

By step 1, Ada source code is analysed and the modifications, needed for activation of Run-Time-Linking, are identified. The relevant information is stored into Add-On-files, from which it is retrieved by Dynamic Configuration Tools. These tools prepare Ada source code for Dynamic Configuration (Step 2). By Step 3 modified Ada source code is compiled and linked together with additional packages for support of Dynamic Configuration.

Step 1 is presently done manually, it will be automated in future. Step 2 is already fully automated. Support packages for step 3 are available.

Ada specific features are not affected like

- omitting of default parameters in procedure calls,
- overloading of procedures.

Other advantages of pre-processing of source code are:

- a software engineer needs not to care about the mechanism of Dynamic Configuration, it is completely hidden - if desired,
- If Dynamic Configuration is not used directly (see section 3.3 below), software can be tested and verified in the usual manner, i.e. using conventional linking, before Dynamic Configuration is activated,
- the decision whether to use Run-Time-Linking or not can be postponed to the end of an implementation phase.

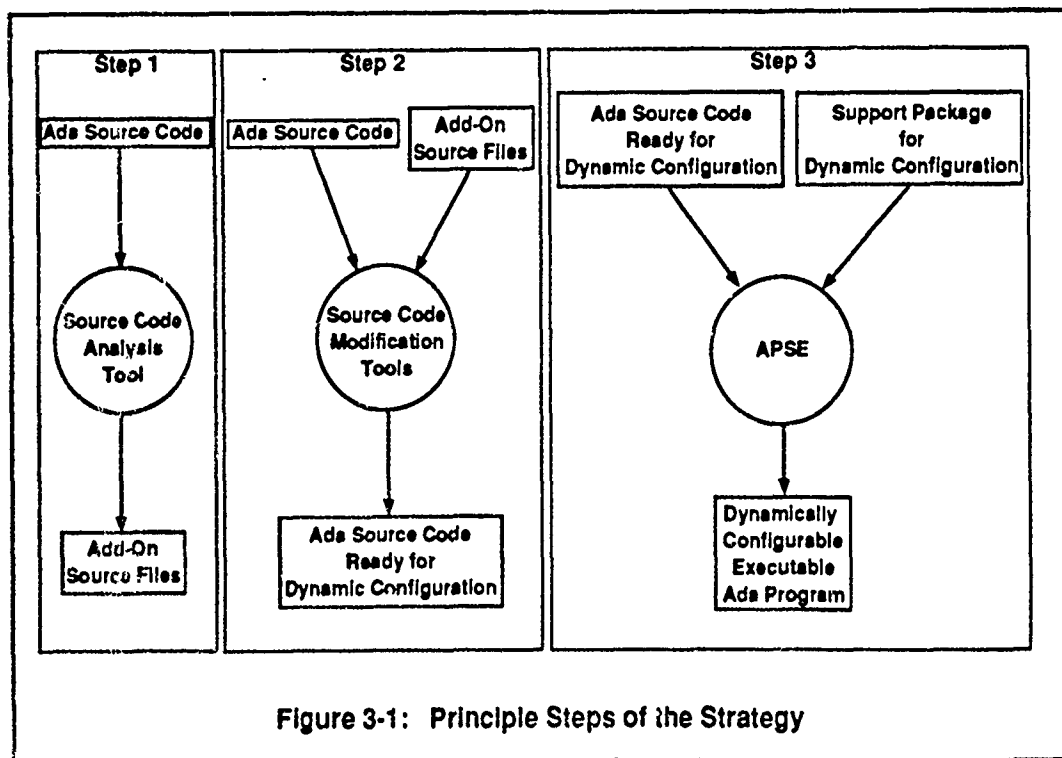


Figure 3-1: Principle Steps of the Strategy

3.2 Implementation of Run-Time-Linking

The addresses of the items (procedures, functions, data) are stored into **Address Tables**, i.e. arrays of addresses (see Fig. 3-2). Each Address Table is describing a certain configuration. The **index**, by which a certain address is retrieved from an Address Table, is representing the **link-specification** (of procedure / function / data) all over the life-time or run-time of a program, an address is representing the **link-target** (actual procedure / function body or memory allocated to data).

In the upper left corner of Fig. 3-2 the process of **on-line update** from version N to N+1 is shown. Addresses of procedure ProcB and data DataD are replaced by addresses of PNewB and DNewD. Before and after update access is still controlled by the fixed indices 2 and 4.

Reconfiguration by **overloading** is shown in upper right corner. For each mode Address Tables are provided each containing mode-specific procedures and data. On-line switch from Address Table 1 to Address Table 2 allows on-line mode transition.

The capability to **call procedures via indices**, i.e. to use data and to directly call procedures accordingly (without using a CASE-statement) is shown in the lower part of Fig. 3-2. Data Set 1, representing a logical program can

be translated into calls of procedures A, B and C, and data set 2 into calls of D, A and E. A specific capability is, that via different indices (2 and 5) the same procedure can be called.

Selection of a data set may be **status driven**, e.g. it may depend on task-id, mode-id and status of a mode. The corresponding logical program is instantiated according to the skeleton defined by the selected data set. This feature allows to instantiate a logical program out of a set of "generic" reusable procedures and data including references to specific procedures and data. It is possible to instantiate a procedure hierarchy including specific procedure calls and data access depending on instantiation parameters. The higher modularisation of software is, the more code is shared and reused between tasks and modes by instantiation of logical programs.

The correspondency between a fixed index and a fixed address allows to use "conventional linking" and "Run-Time-Linking" in parallel during development.

Furthermore, the Dornier specific implementation of Run-Time-Linking guarantees full integrity of a program at run-time

- in a pre-emptive real-time environment, and
- in case of multiple, interdependent changes in software.

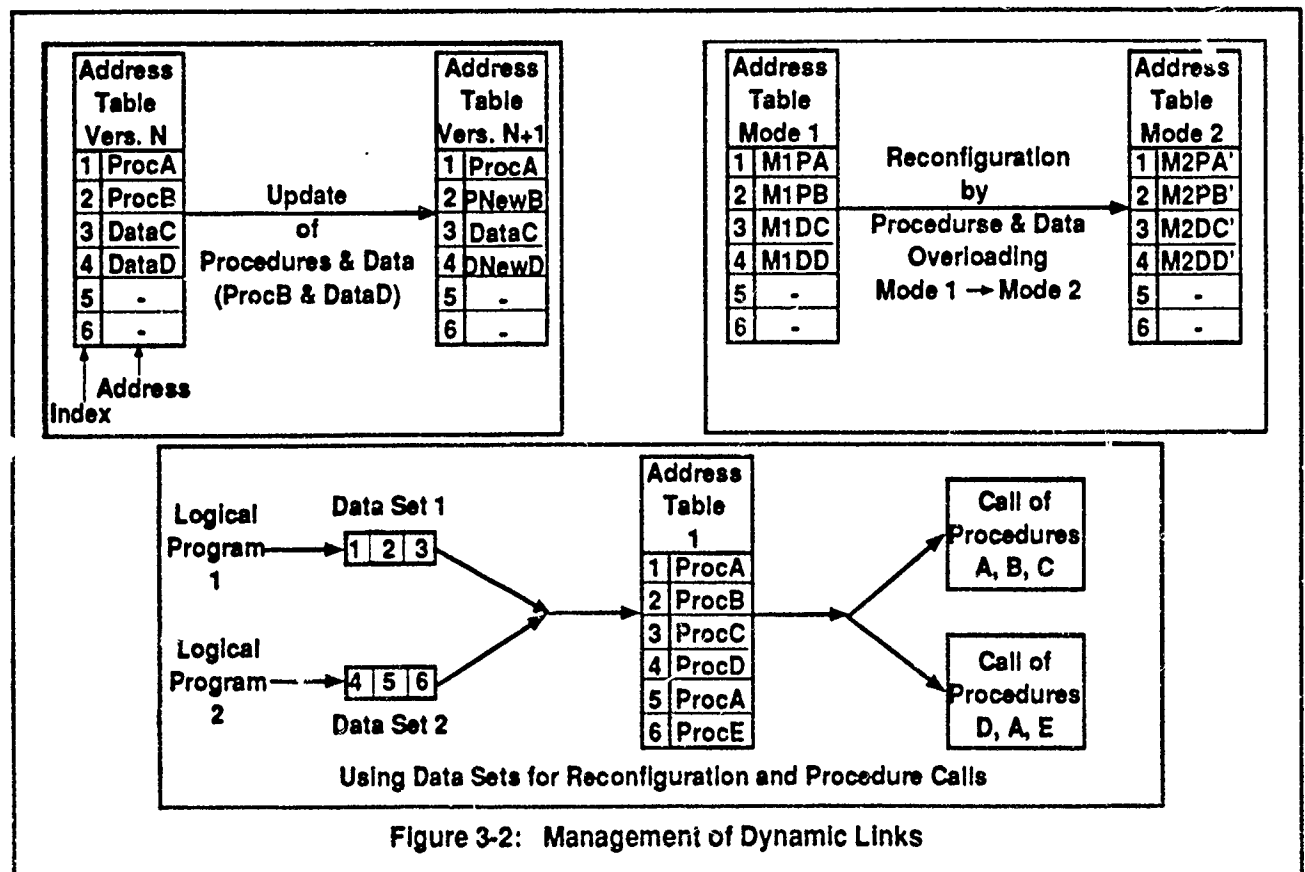


Figure 3-2: Management of Dynamic Links

3.3. Possibilities to Use Dynamic Configuration

Dynamic Configuration can be used twofold: **directly** and **indirectly**.

Direct use of Dynamic Configuration, e.g. for temporary, logical instantiation of a task body, means explicit use of access via Address Tables, i.e. it implies direct use of the entries (indices) into an Address Table. In this case no equivalent conventional Ada source code exists, steps 1 and 2 in Fig. 3-1 are not needed for generation of executable code. However, for verification purposes step 1 and 2 are still needed.

Indirect use of Dynamic Configuration means modification of existing source code in order to get the desired dynamic capability. This allows to develop programs in the usual manner.

3.4. Realisation of Reconfiguration and Code Update

In case of on-line update two executing programs exist: the old and the new version (previous and next version), running on two different processors simultaneously: the task of on-line code update is to adapt the executing old, previous version to the new, next version without disturbing and corrupting the executing parts. This requires careful consideration of the program environments.

In case of on-line reconfiguration only one program is executing, and its logical structure is modified via Run-Time-Linking.

3.4.1. Operational Scenario for Code Update

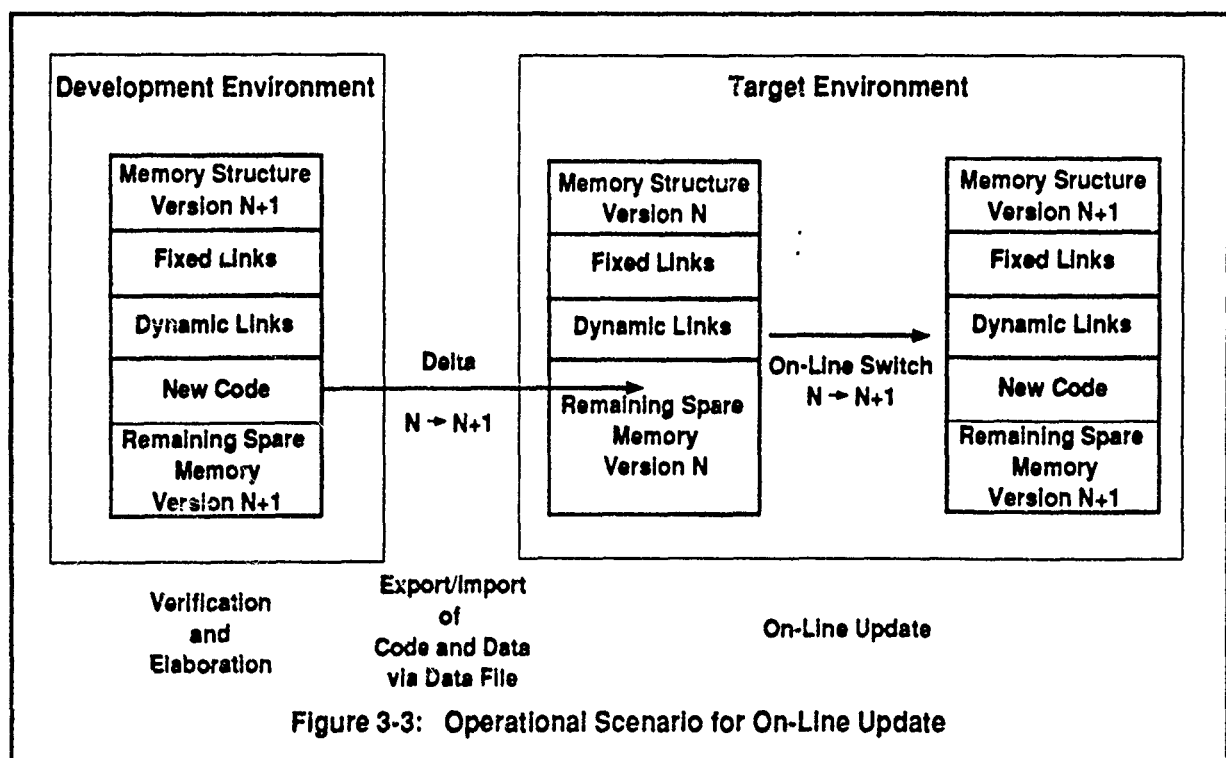
Two principle environments exist in case of code update due to software maintenance: a **Development Environment** and a **Target Environment**. Software updates or new software are developed on the Development Environment and then loaded onto Target Environment to become operational.

We can classify the updated program into two parts: one part, which is **not modified**, and another part which is **completely new** or is **updated**. Now, the basic idea (see Fig. 3-3) for on-line update is, to keep during on-line update the old program version running in that part, which is not affected by the update, and to exchange just those parts which are to be modified or which have to be added or removed.

Ensuring that the new executable code, which is imported from Development Environment, fits with the address structure of the (executing) program on Target Environment, the imported code can be activated without recompilation and relinking of the (executing) program on Target Environment.

Of course, spare memory for imported code and data have to be reserved on the Target Environment.

Run-Time-Linking provides the means to activate software which was not known at link-time of the running program and allows to reallocate code and data at run-time. Memory management procedures support allocation of spare memory and release of inactive memory areas.



The essential point for software update is to ensure compatibility between the executable code of Development and of Target Environment, after having changed the program on Development Environment. Automated procedures will be provided to ensure the needed compatibility.

Compatibility does not mean that symbol addresses on both environments must be exactly the same. Only those addresses must be identical, which are not accessed via Run-Time-Linking. All other addresses may be different.

The software engineering strategy and associated tools must guarantee the needed compatibility of addresses. These tools may be considered as compiler Add-On's.

3.4.2. Operational Scenario for Reconfiguration

Reconfiguration means instantiation of another logical program either by (see Fig. 3-2)

1. exchange of addresses for a certain entry in an Address Table,
2. activation of another Address Table, providing another address for the same entry, or
3. using another data set specifying the desired logical program.

As reconfiguration affects one program only, compatibility of addresses must not be considered.

A mode transition is a specific case of reconfiguration. During a mode transition

- the (old) operational mode has still to be executed until mode switch is performed,
- the next mode has to be prepared by additional activities, in parallel to the operational step sequence,
- a switch has to be performed when the new mode can be entered: the new mode becomes operational,
- the previous (old) mode has to be deactivated by additional activities, executed in parallel to the (new) operational step sequence.

Using Dynamic Configuration, the operational procedures for a mode transition become very simple.ⁱⁱⁱ

First, one program can be used for all tasks. Therefore a mode switch can be (synchronized) by this program only as it holds all relevant information.

Second, switch from one configuration to another one can be done immediately within a few microseconds, just the time needed to change a configuration

parameter.

Third, the additional activation / deactivation steps can easily be added to the operational steps, although two different (logical) programs are executed immediately one after the other: before mode switch the old mode in status "operational", the new mode in status "activation", after mode switch the new mode in status "operational" and the old mode in status "deactivation". Dynamic Configuration allows to separate logically the activities of both modes, but to execute them immediately one after the other according to a predefined (simple) scheduling scheme including synchronisation - for all task frequencies.

The logical switch from one Address Table to another is just equivalent to a physical switch from one memory bank to another, each containing the relevant program.

3.5. Dynamic Configuration and Ada Language

Some aspects concerning Ada language are discussed now in order to make clear that the concept is compliant with Ada philosophy.

3.5.1. Interface Verification

In case of procedure / function calls assembler routines establish the dynamic link. Usually, verification checks are suppressed for a transition Ada - assembler. However, Dornier's concept for Dynamic Configuration provides the capability for interface verification at this transition. This is an option, which can be suppressed - like the checking capabilities of an Ada compiler - in order to get better performance.

3.5.2. Elaboration

Importing executable code into a running program requires that it has already been elaborated.

This Ada rule is completely fulfilled, because all code is elaborated on the Development Environment before it is imported into Target Environment. As the programs on the Development Environment and on the Target Environment are compatible, it makes no difference where code is elaborated.

3.5.3. Visibility Rules

Although global Address Tables are used, information hiding is provided like in pure Ada: only procedures, functions and data can be accessed, which are known according to the WITH-Hierarchy.

3.5.4. Features Needed from Chapter 13

From an APSE following non-standard Ada features are needed:

- package System
- interface to assembler, i.e.
 - package machine_code,
 - capability to link assembler object files with Ada code or
 - other equivalent capabilities,

ⁱⁱⁱFor the following consideration it is assumed, that several (at least one) cyclic tasks with different cycle periods are running and that the task bodies are mode-dependent.

- capability to affect memory allocation in order to get a compatible memory structure.

- b. using different logical names but to access the same physical data area in memory.

4. Capabilities

4.1. Real-Time Capabilities

The concept is tailored for preemptive real-time environments. The switch to another logical program, procedure, function, data area is sufficiently simple and therefore can easily be protected against preemption.

An essential feature for real-time processing is the capability to immediately instantiate a logical program. The consequence is that the same program can be used for all tasks and all modes. Therefore tasks have only to be created for each frequency, - using the same Ada program - but not for each combination of tasks and modes.

4.2. Overloading Capabilities

Beside on-line reconfiguration and program update other features are provided:

1. Overloading of procedures / functions:

In Ada overloading for procedures / functions is supported, if procedures of same name have different parameter lists. By Dynamic Configuration overloading is possible for procedures/functions which have identical parameter lists. This is a feature which is used for mode management. E.g. the activities for a control-law-processing task can be characterised by

- read sensor data
- perform control-law-calculation
- send actuator commands
- update state matrix.

in this sequence, steps are identical for each mode from a logical point of view, but - of course - the body of each step is different. It is very easy to perform a mode transition using Dynamic Configuration in this case. What has to be done is just to use the same logical sequence and to exchange the mode-dependent procedure bodies by on-line reconfiguration (see upper right corner of Fig. 3-2).

2. Overloading of data:

Overloading of data is not supported by Ada. However, Dynamic Configuration allows two principle kinds of data overloading:

- a. using the same logical name but to access different physical data areas (of same structure, of course) at different times.

All capabilities are available by a demo implementation.

5. Experience

5.1. Activities in the Past

Activities on this subject of Dynamic Configuration were started in 1986, when the Technical Center (ESTEC) of European Space Agency (ESA) requested a concept for on-line update of code in view of time-critical and/or autonomous space missions. A first idea was borne and implemented in C. Then it was investigated (in 1989) if the concept could be ported to Ada, as Ada is the programming language for future space missions.

To demonstrate the feasibility of the concept with Ada, Dornier defined the needed software engineering strategy and implemented a demo. This demo was presented on the EUROSPACE Symposium in Barcelona¹ in December 1990 and demonstrated on-line reconfiguration and update capabilities on a PC using Meridian Ada compiler.

Then the demo was enhanced to demonstrate fully the capabilities of the concept for task management and mode transitions. These features were presented on ESA's First International Conference on Spacecraft Guidance, Navigation and Control in June 1991.²

The next goal was to demonstrate the portability of the concept. The Alsys compiler was selected for this step, as it is foreseen as target compiler for the European space projects COLUMBUS and Ariane V and for NASA's space station FREEDOM.

Tools already exist to make Ada source code ready for Dynamic Configuration. The toolset will be completed in near future in order to fully automate the preparation procedure. Support packages for Run-Time-Linking and management of code import/export are also available.

Since its first implementation in 1990 the concept itself was improved and constraints - initially existing - could be removed. Performance test programs were used in order to analyse performance of each compiler and to investigate the best algorithms for Dynamic Configuration on each APSE.

Dynamic Configuration with Ada was implemented on top of APSE's without needing compiler-internal details from vendors. Furthermore, it was possible to achieve compatibility of memory allocation between Development and Target Environment on a PC-DOS environment, which basically does not support user-controlled memory allocation at all.

5.2. The Demo

The demo is presently available for a PC-DOS-environment for Meridian (V 4.01) and Alsys (V 4.4.1) compiler. It provides multi-tasking and mode switching capability together with other features specific for Run-Time-Linking (described in section 4.2).

It supports priority-based, quasi-preemption^{iv} of tasks under DOS using only the DOS-clock and no interrupts. "Quasi-preemption" means that tasks can be preempted at predefined logical breakpoints, but not at arbitrary times²: the next logical step is executed only, if there is sufficient time to finish it before the next task of higher priority is started.

Activation and deactivation of modes in "parallel" to execution of the actual operational mode is implemented. By parameters, the additional workload during a mode transition, caused by activation/deactivation steps, can be adjusted by the user. Of course, immediate, unconditional mode switch is possible.

Priorities can arbitrarily be assigned to tasks - independently from their execution rate - and can be changed on-line during execution - if desired.

The demo application consists of three tasks:

1. the command interpreter,
2. a task to export/import updated code, and
3. a task for control-law processing.

The control-law-processing task is used to demonstrate on-line reconfiguration / mode transition and code update.

The application program is available in a version on Development Environment ("on-ground") and on Target Environment ("on-board"). The on-ground version contains the new code and data and exports it via a data-file (see Fig. 3-3). The on-board version imports the data file and stores its contents into a predefined spare area. Of course, new code and data are not available in the on-board version before the update-file is imported. The update-file can be modified by a debugger and after modification directly imported into the executing program to demonstrate that new code is really imported. The impact of a file-change is immediately visible, e.g. RETF can be inserted in the data file at the beginning of a procedure resulting a "NOP". When the relevant procedure is executed, no output is generated on screen due the immediate RETURN in the procedure.

5.3. Portability

When talking about portability of the concept and the implementation, we have to consider different parts which are subject of portability:

- the Software Engineering Concept

- the assembler routines and
- the memory allocation strategy interfacing with an APSE.

The Software Engineering Concept is fully portable, because it is based on standard Ada. Modification towards Dynamic Configuration is supported by tools on source code level, so that it is easy to port an implementation.

The assembler routines and memory allocation strategy for on-line update is not portable, in general. However, this is not a real problem of the user, because the needed support will be provided for each APSE.

6. Performance

6.1. Performance Analysis

Implementation of the demo was the first step to demonstrate the feasibility of Dornier's concept. The next step was to improve performance.

We found that performance of certain algorithms depends on the compiler. A certain implementation can give optimum performance on one APSE, but worse performance on another APSE. E.g. on the first APSE we changed our initial implementation and could increase timing performance by about a factor of 10. On the next APSE the second approach was slightly worse than the initial one, but both about a factor of 10 faster.

By consequent performance analysis and improvement we have achieved now a mature implementation for both APSE's.

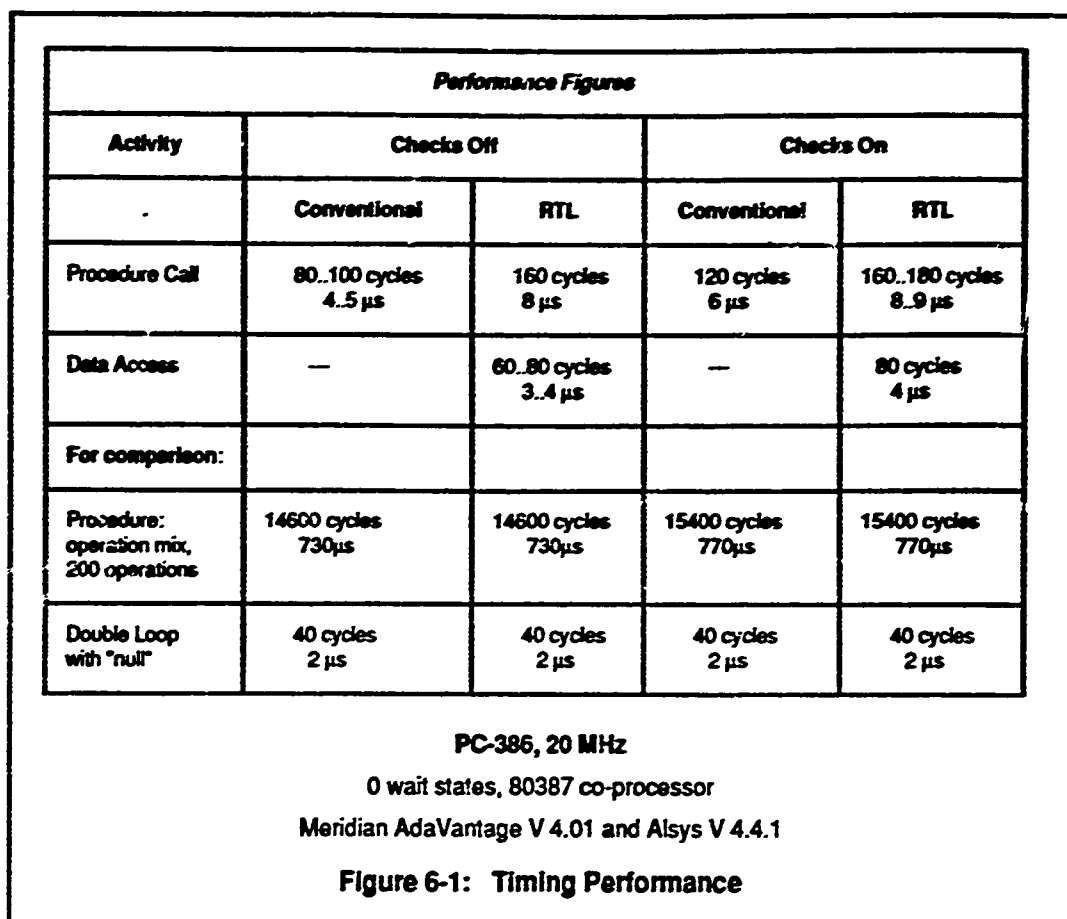
6.2. Results

Figure 6-1 gives some figures for Meridian and Alsys compiler for procedure and data access and two other examples for comparison. Obviously, no significant difference in timing performance occurs for procedure calls and data access, because the figures are derived from the optimum implementation on each APSE: the implemented algorithms are slightly different.

The figures for "procedure call" are related to a call of a dummy procedure with two parameters, containing no data declarations and no executable code. The overhead of 2 .. 3 μ s has to be compared with time needed for execution of code in a procedure like a mix of assign statements consisting of boolean (10%), character (10%), byte (10%), integer (10%), long_integer (4 bytes) (20%) and float (8 bytes) (40%) operations - given for comparison.

For a procedure call about 8 bytes have to be pushed on the stack in addition. Inside a procedure, for each data structure accessed via Run-Time-Linking, about 4 bytes are needed on stack plus code for initialisation of the link.

^{iv}This is an outcome of the concept, but not a precondition to use it



7. Conclusions

It has been pointed out that the concept for Dynamic Configuration with Ada

- is compliant with Ada's software engineering philosophy,
- increases reliability of a system due to the possibility to use simpler procedures for reconfiguration / management of mode transitions and code update,
- is sufficiently portable among hardware and software platforms,
- is user-friendly by provision of guidelines and automated procedures,
- has sufficient performance,
- is designed for use in a preemptive real-time environment.

The concept is now mature for safety and time-critical systems either for new projects or for improvement of existing software.

REFERENCES

1. R. Gerlich 1990, On-Line Replacement & Reconfiguration of Ada Real-Time Software, **First EUROSPACE Symposium on "Ada in Aerospace"**, Barcelona, Spain, 254-273
2. R. Gerlich 1991, Mode Management and Run-Time-Linking with Ada in Real-Time, **First ESA International Conference on "Spacecraft Guidance, Navigation and Control"**, Noordwijk, The Netherlands, to be issued

Author's address: Rainer Gerlich, Dornier GmbH, Space Electronics Division, Dept. TED, P.O.Box 1420, D-7990 Friedrichshafen, Germany (W), Phone +49/7545/8-2124, Fax +49/7545/8-4411

About the author: Rainer Gerlich is senior engineer of software engineering for space systems. His research interests have included database software, image processing, and formula manipulation in AI. His current research interests include architecture of real-time systems, software methodologies and CASE, standardisation and reuse of software, adaptive systems.

HANDLING PRIORITY INVERSION PROBLEMS ARISING DURING ELABORATION IN ADA PROGRAMS FOR REAL-TIME APPLICATIONS

Leslie C. Lander and Sandeep Mitra

Department of Computer Science

The Thomas J. Watson School of Engineering, Applied Science and Technology
State University of New York, Binghamton, NY 13902-6000

Abstract—Priority inversion is a recognized and serious problem for real-time systems. One form of priority inversion is observed among Ada tasks at program "start-up time" and arises due to the *elaboration order* (of the packages constituting the Ada program) chosen by the compiler. This paper reports on how such priority inversion can give rise to errors in otherwise sound Ada programs; errors that are highly unlikely to be foreseen by the programmer and whose cause may be difficult to diagnose. The impact on the Ada implementation template for Rate-Monotonic Scheduling is presented as an example. Perceived deficiencies in Ada semantics with respect to priority inversion during elaboration are discussed and various viable solutions are presented.

Keywords: elaboration order, priority ceiling protocol, priority inversion, rate monotonic scheduling

Introduction

Priority inversion has been defined^{7,14} as any situation where low priority tasks are served before higher priority tasks. These papers, as well as Reference 11, report various forms of priority inversion that may repeatedly occur during the execution of an Ada program due to the FIFO nature of entry queues and the scheduling of selective wait options. *Priority inversion during elaboration* (PIDE), occurring during the elaboration phase of a program (thus, at "start-up time") has been reported in detail in previous papers.^{10,12} The reason for the existence of such priority inversion can be found in the LRM,¹ Sec. 9.3, where it is stated that a task object not created by an allocator must begin activation after the elaboration of the declarative part within which the task object occurs immediately. Thus, in effect, task activation is required to commence just after passing the reserved word *begin* following the declarative part, and assuming an implicit *begin* if none exists. As reported elsewhere,^{4,6,20} such tasks then begin to execute

concurrently with the statements of the enclosing unit. Consequently, tasks that occur immediately within library packages begin to execute concurrently with the execution of the initialization statements of these packages (*viz.*, the sequence of statements that occur after the reserved word *begin* following the declarative part of the package body). Now note that, in the case of a library package, the initialization statements execute at elaboration time. Therefore, the order of execution of these statement sequences is the order in which the library packages are elaborated. From the above observations, it is evident that tasks enclosed by library packages commence execution in an order which will differ across various Ada compilers, since the rules determining the order in which program units must be elaborated are very loose (LRM, Section 10.5). Besides, assigned priorities of such enclosed tasks have no influence on the elaboration order, since the LRM makes no mention of the role of priorities of such tasks in determining elaboration order. The potential for PIDE is thus very definitely extant in current Ada, and we have observed it occurring in several compilers that we have studied. As a simple example, consider the package specifications shown in Figure 1. The main procedure and package bodies are not included for the sake of clarity.

If the body of package P2 is elaborated before the body of package P1, then the initial order in which the enclosed tasks begin to run will be P2.T2_pr2, P2.T1_pr1, P1.T2_pr10, P1.T1_pr4.* Priority inversion is evident. Though, as we stated earlier, such priority inversion

* We assume that the main program has the lowest priority available. For the Ada systems we have tested, the behaviors reported and the coding practice suggested are valid only if such is the case.

occurs only at "start-up time," failure to recognize that it may occur can prevent an otherwise correct program from even beginning to execute. The impact of PIDE on the implementation, using Ada tasks, of a real-time system paradigm is described in detail in the following section. The paradigm that we have used is the Ada implementation template of Rate Monotonic (RM) Scheduling, including the Priority Ceiling Protocol (PCP), recommended by Sha and Goodenough¹⁷ and by Borger, Klein and Veltre.⁵

```
package P1 is
  task T1_pr4 is
    pragma priority (4);
  end;
  task T2_pr10 is
    pragma priority (10);
  end;
end P1;

package P2 is
  task T1_pr1 is
    pragma priority (1);
  end;
  task T2_pr2 is
    pragma priority (2);
  end;
end P2;
```

Figure 1. Tasks which are susceptible to priority inversion due to elaboration order.

Impact of PIDE on a Real-Time Implementation

Consider the implementation, using Ada tasks in a uniprocessor environment, of the RM Scheduling theory, including the PCP, as described by Sha and Goodenough.^{9,17,18} We summarize briefly the salient features of RM and PCP below. The tasking model consists of *periodic tasks* communicating through *binary semaphores*. The period of each task is fixed, and let us assume that it also indicates the *deadline* of the task. The execution of each iteration of the main body of the task must complete before its deadline. Each iteration becomes ready-to-run at the start of the period. RM theory states that the lower the period of the task, the higher its priority. PCP assigns a *ceiling* to each semaphore, which is the highest task priority amongst all tasks that use the semaphore. Despite the fact that

high priority tasks can preempt lower priority tasks, PCP does not allow a task to enter its *critical region* unless its own priority is higher than the ceilings of all semaphores currently locked by other tasks. Thus, a low priority task T_{low} can block a high priority task T_{high} , with priority p , when the situation is such that T_{high} is trying to enter its critical region and T_{low} currently holds the lock on a semaphore S that T_{high} , or some other task with priority greater than or equal to p , uses. This is the only way in which a low priority task can block a higher priority task. Also, PCP ensures that any task T will not be allowed to enter its critical region unless it will not request to lock any semaphore that is currently locked by any other preempted task T' and hence a deadlock between T and T' cannot occur.

Sha and Goodenough and Borger *et al.*, suggest an Ada implementation template, with appropriate design approaches and coding practices in Ada to adequately implement RM Scheduling with PCP. The basic idea of their template is to implement the semaphores, which are the sole means through which other "client" (periodic) tasks communicate, by using "monitor" Ada tasks. When a client wishes to enter a critical region, it makes a task entry call to an entry of the appropriate monitor task. The principal code structure in a monitor task is a selective wait, enclosing the accept statements corresponding to all its entries. Each client of the monitor calls its own entry, and no others. When a call from a client task is accepted by the monitor, the code executed during the rendezvous (by the monitor) is actually the critical region of the client task being executed by the monitor *on behalf of* the client. Borger *et al.*, present four models for the implementation of periodic tasks in Ada, some of which are the *Delay* model, *Task Dispatcher* model and *Delay_Until* model. An example of this implementation template, using the *Delay* model, is provided in Figure 2, which is supposed to actually implement the following behavior:

Consider two periodic tasks $T1$ and $T2$. In addition, there are two binary semaphores $S1$ and $S2$, used by both tasks. Suppose $T1$ locks the semaphores in the order $S1$, $S2$, while $T2$ locks them in reverse order. Further, assume $T1$ has a higher priority than $T2$. Thus,

T1 : {...P(S1)...P(S2)...V(S2)...V(S1)...}
 T2 : {...P(S2)...P(S1)...V(S1)...V(S2)...}

According to PCP, since both semaphores are used by T1 and T2, neither task will be allowed to enter any critical region if the other holds the lock on a semaphore. Thus, deadlocks will be prevented. The Ada code skeleton in Figure 2 is expected to model this behavior; P1.T corresponds to T1, P2.T to T2, P3.Sem to S1 and P4.Sem to S2. Sha and Goodenough have reported that, in order to correctly implement PCP, it is necessary to eliminate, amongst Ada tasks in the above template, priority inversion that arises on account of FIFO entry queues and selective waits. To achieve this effect using program structure rather than a modified run-time support (RTS), they have suggested that monitor tasks be given a priority *one level higher* than the maximum priority of all their clients. Also, they have stated that monitor tasks should not be suspended during rendezvous. Thus, a monitor task is always ready to rendezvous with any of its clients. Note that in Figure 2 the monitor tasks have been given a priority higher than their clients and our Ada RTS is such that they are not suspended during rendezvous.

A particular Ada compiler may use the context clauses to determine the following elaboration order of package bodies: P1, P2, P3, P4. Note that this order may not be unique and different compilers may compute different orders; the behavior that we describe below is therefore also not unique and may vary over compilers, thus leading us to note that there is a certain unpredictability introduced into program behavior as a result of varying elaboration orders. Programs become non-portable as a result—the correct behavior observed after having carried out testing using a certain compiler may not be repeated in another.

First-entry assumption: we require that a fixed choice be made by the Ada RTS when a selective wait is executed for the *first* time in a task during program execution. We shall assume that, during the first execution cycle, the entry which is being called and which appears *earliest lexically* in the task specification, is necessarily accepted. The Ada compilers we have used do make *some* arbitrary choice of this kind.

With the above assumptions about the elaboration order and choice of selective wait, the blocks of code labeled in this example are first executed as follows:

P1.T(A), P2.T(A), P3.Sem(C), P4.Sem(C), DEADLOCK

When P3.Sem and P4.Sem accept their entries labeled E1, they execute within the accept blocks the critical regions on behalf of P1.T and P2.T respectively. Each accept block contains a nested entry call. If the higher priority tasks P3.Sem and P4.Sem began to execute earlier, as expected, then deadlocks would be precluded as shown for such templates.¹⁷ However, P1.T and P2.T are activated and begin to execute earlier, as a result of the assumed elaboration order. Note that it is assumed that the first delay statement in the body of P1.T and P2.T does not block these tasks for long enough to prevent their execution from proceeding before P3 and P4 bodies are elaborated; such may well be the case if elaboration times are long enough (e.g., due to a very long initialization procedure). Both P1.T and P2.T are blocked when first trying to enter their critical region though such blocking is not sanctioned by PCP and is the result of P3.Sem and P4.Sem having not begun to execute despite their higher priorities. When P3.Sem becomes ready to run upon activation and executes, there is an entry call waiting and when P4.Sem executes it also has entry calls pending; in fact, it has two. A deadlock occurs now because P4.Sem accepts the call to entry E1, by the first-entry assumption above, resulting in a nested entry call to P3.Sem which is blocked on an entry call to P4.Sem. Note that the deadlock occurs despite the priority profile of the participating tasks, which is expected to model the behavior of the PCP, and therefore should preclude tasks waiting at one or more different entries.

The sole reason for the abnormal behavior above arising in the template despite giving monitor tasks higher priority is that the effect of higher priority is nullified by PIDE. As a result client tasks begin to perform "useful work" earlier than the monitor tasks they call. The activation rules for Ada tasks are such that, in a uniprocessor environment, the assigned priorities may be respected in the order of activation only if the concerned

```

package P1 is
  task T is
    pragma priority(2);
  end T;
end P1;

package P2 is
  task T is
    pragma priority(1);
  end T;
end P2;

package P3 is
  task Sem is
    entry E1;
    entry E2;
    pragma PRIORITY (3);
  end;
end P3;

package P4 is
  task Sem is
    entry E1;
    entry E2;
    pragma PRIORITY (3);
  end;
end P4;

-- context clauses ...
package body P1 is
  task body T is
  begin
    delay (Task_Start_Time - Calendar.Clock);
    loop
      ...block of code (A)
      P3.Sem.E1;
      ...block of code (B)
    end loop;
  end T;
end P1;

-- context clauses ...
package body P2 is
  task body T is
  begin
    delay (Task_Start_Time - Calendar.Clock);
    loop
      ...block of code (A)
      P4.Sem.E1;
      ...block of code (B)
    end loop;
  end T;
end P2;

-- context clauses ...
package body P3 is
  task body Sem is
  begin
    loop
      select
        accept E1 do
          ...block of code (C)
          P4.Sem.E2;
          ...block of code (D)
        end E1;
      or
        accept E2 do
          ...block of code (E)
        end E2;
      end select;
    end loop;
  end Sem;
end P3;

-- context clauses ...
package body P4 is
  task body Sem is
  begin
    loop
      select
        accept E1 do
          ...block of code (C)
          P3.Sem.E2;
          ...block of code (D)
        end E1;
      or
        accept E2 do
          ...block of code (E)
        end E2;
      end select;
    end loop;
  end Sem;
end P4;

-- context clauses ...
procedure Main_Pgm is
  ...
begin
  ...;
end Main_Pgm;

```

Figure 2. An implementation template of the Priority Ceiling Protocol.

tasks are in the same enclosing compilation unit. Thus, an obvious solution to the problem would be to require all tasks to be enclosed in the same unit. However, this is too restrictive and more viable options are discussed in a later section. In this context, note that using the *Task Dispatcher* model, suggested by Borger *et al.*,⁵ (also used by Vestal¹⁹) is also over-centralized and restrictive. This approach supposes a central Dispatcher task that makes all clients ready to perform "useful work." Also, a `TIMER_INTERRUPT` from a timer initialized by the main program is used, assuming that the main program begins to execute after all package bodies are elaborated and that all library tasks have begun execution. This model serves the purpose that all monitor tasks will be ready to rendezvous whenever they are called, thus nullifying the adverse effects of PIDE for this template. Its restrictive nature is illustrated by the fact that the Dispatcher must know the identities of *all* client tasks, each of which must be given an entry `DISPATCH` for the Dispatcher to call. This structure prevents client tasks from being entirely declared inside library package bodies. Besides, this model requires the user to explicitly manage the scheduling of tasks; ideally, this function should be abstracted out of the application program.⁵

Using the pragma `ELABORATE` to force earlier elaboration of package bodies enclosing monitor tasks is also not always a workable option, for the dependencies amongst initialization procedures may require the pragma to be used on certain program units. Such placement of the pragma may well be inconsistent with the needs of the early elaboration of the packages containing monitor tasks. Additional deficiencies of pragma `ELABORATE`³ are beyond the scope of this paper.

Other models for the implementation of periodic (client) tasks provided by Borger *et al.*, are also not adequate to counter the effects of PIDE occurring in the above template. These models are the *Delay* model, which uses the Ada delay statement as illustrated in Figure 2 and the *Delay_Until* model, which assumes the existence of a `delay_until(Absolute_Task_Start_Time)` statement. Both these approaches require the user to specify an absolute start time for the client tasks, which may not be the same for all such tasks. Assuming that system execution

commences when the value of the absolute time is zero, the absolute value of the starting time for each client task must be computed in such a manner so as to ensure that the task begins to perform "useful work" *only after* all packages enclosing the monitor tasks called by the client task have been elaborated. Otherwise, the client task may request a rendezvous with a monitor that has not yet begun to execute, resulting in a deadlock as described above. Accurate computation of the start time is therefore indispensable and in order to achieve this goal, real-time Ada compilers would be required to provide precise estimates of cumulative elaboration times of certain sets of packages. Firstly, on account of the absence of a unique elaboration order, this may well be different for different implementations. Secondly, it may also be impossible to obtain accurate estimates of the total elaboration time, as the initialization procedures may use input data. The use of optional optimizations may also have an effect.

Although we have used the well-known RM Scheduling Theory to illustrate the problem of PIDE, it came to our attention through its impact on our other research.¹⁵ We are working on the development of a methodology to automatically map specifications drawn up using a state-oriented, real-time formal technique (SAN)¹⁶ into Ada implementation code. Proper mapping of the semantics of SAN systems requires extensive initialization of many Ada objects generated by the mapping algorithm, thus rendering the total elaboration time significant with respect to the execution times.

In subsequent sections, we suggest a coding practice, a pragma, and a change to the language standard, any of which eliminates the problem of PIDE for any Ada program with prioritized tasks.

Classification and Activation Rules for Tasks

As mentioned in the introduction, PIDE arises due to the activation rules for tasks. PIDE can be considered true to Ada semantics if the affected high priority tasks can be considered *ineligible for execution* (LRM, paragraph 9.8(4)) as the enclosing package body has not yet been elaborated. Cohen⁶ states that a task is "ready to execute" if it is not completed, it is not blocked, and

there are sufficient computing resources available. It is not evident, however, that a task object whose specification has been elaborated, but which has not yet been activated, cannot be considered ready to run. We have already seen that the activation rules for library tasks result in the commencement of their execution during the elaboration phase of the program. These tasks are however not required to terminate. The rationale behind this asymmetry is stated in the revision issue RI-2016:³ the automatic termination of library tasks when the main program completes is fundamentally at odds with the paradigm, frequently used in real-time systems, which uses a "vacated" main program and does all the work in library tasks. It seems evident that the possibility of having the library tasks constitute the set of all actors in a real-time system was intentionally included by the language designers. The ill-structured way in which tasks are activated during elaboration may prevent this set of actors from starting to execute properly and we now suggest a model to rectify these problems.

We propose a classification strategy for tasks based on the method of creation and the nature of the direct masters of tasks.

1. A task that is not created by an allocator and whose direct master is a library package is classified as a *pervasive* task.
2. A task that is not created by an allocator and whose direct master is a task, block statement or subprogram is classified as a *non-pervasive* task.
3. A task created by an allocator is classified as an *anomalous* task, as such a task depends directly on the unit that elaborates the corresponding access definition and not on the entity whose execution creates the task.

Since non-pervasive tasks exist only in the execution lifetime of the ephemeral entity (subprograms, block statements and tasks, all of which can complete execution and/or terminate), the current activation and termination rules are somewhat appropriate for them. However, it may be noted here that PIDE may well occur

amongst a set of non-pervasive tasks (e.g., tasks declared within subpackages in the same procedure body) under the current activation rules and such priority inversion can be prevented by our recommendations below. For anomalous tasks the current activation rules may also be considered appropriate for it is assumed that such tasks will be created only in the event that we do not know how many tasks we shall eventually need, or if we need to exchange task identities at some point during elaboration. As an aside, note that we may, however create an anomalous task that is not terminated though the entity that created it has completed its execution and no communication is possible with this task because the scope of its identifying object has been exited. Such tasks may well prove to be potential sources for deadlock.

We now define a rule that should eliminate PIDE amongst pervasive tasks. We assert that pervasive tasks should be activated *after* the elaboration of the declarative part of the *main program* and just after the reserved word *begin*. If such is the case, then the Ada RTS will ensure that no priority inversion takes place when pervasive tasks start to execute. This will in fact be required by current Ada rules that state that the activation of tasks proceeds "in parallel" (LRM 9.3). The consequences of this new rule are discussed in Lander *et al.*¹³ Barnes⁴ discusses the structure summary and main program of a typical Ada system, and states that a model of the "complete" program can be understood in terms of one where the package STANDARD is declared inside a block statement that appears in the body of an environment task, and all library packages and package bodies can be considered to be declared as subpackages immediately within the package body STANDARD. These library packages and package bodies appear in an order consistent with the context clauses and the pragma ELABORATE. The main program is called as one of the "initialization" statements of package body STANDARD. Barnes states that this model captures the fact that tasks that depend on a library package (and are not designated by an access value) are started at the end of the declarative part of STANDARD and before the main program is called. If this were true, then PIDE would certainly, according to current Ada rules, be precluded. However, we have observed that this is not the case in several Ada compilers, primarily due to the activation

rules which, by stating that activation begins after the elaboration of the declarative part within which the task object occurs *immediately*, appear to be making activation determined by the *enclosing lexical entity* and not the master of the task. As we have stated earlier in this paper, many authors of Ada textbooks state that, after activation, the initial task execution commences concurrently with the enclosing entity, though this is not explicitly stated in the LRM. If the task under consideration is a non-pervasive or anomalous task, this rule, as we have seen, makes some sense, but not for pervasive tasks that are not required to terminate (since library packages do not really "complete their execution and get exited from or terminate" but simply complete initialization—a one-time event). Therefore, Barnes' perception about the starting point for the execution of pervasive tasks is appropriate though not universally implemented. The lack of explicit instructions in the LRM on commencement of execution and non-orthogonality in Ada, whereby the termination, but not the activation, of a task is related to its master, is the root cause of this problem.

Through our classification strategy and suggested activation rule, we are making a case for the activation of a task to be related to its master, and also suggesting a new definition of master for pervasive tasks. Barnes states that a package is merely a passive scope wall and has no dynamic life. This reinforces our argument that the execution of the initialization statements in a package body is qualitatively different from the execution of a task, subprogram or block statement. Pervasive tasks should therefore depend on the environment task and not the enclosing package, and this should be made clear in the Ada 9X LRM. In the Ada 9X Mapping Document,² no mention is made about any modification to the activation/termination rules for tasks and the definition of the master of a task. A serious consequence of changing the activation rules is the increased possibility of "elaboration-time deadlock." A discussion on this form of deadlock is presented in Lander *et al.*¹³

It would also be advisable at this point to note that as a result of current activation rules pervasive tasks activate and begin to execute in an ill-structured manner, *viz.*, they do not all become ready to run at the same time.

Worst-case analysis of RM Scheduling and the compile time scheduling of an Ada subset⁸ require all tasks to start at the same time. The current activation rules may hinder such modelling.

Avoidance of PIDE

The modifications to the Ada language standard, discussed above, are a possible means of avoiding PIDE. However, within the framework of the current standard and even the Ada 9X proposals, the following two methods are entirely satisfactory.

Coding practice.

We assume that in current Ada programs, irrespective of the order of activation, it would sufficiently satisfy the user's needs if all pervasive tasks began to perform useful work in the order of their assigned priorities. To achieve this effect, require all pervasive tasks to make an entry call to a common task as their first executable instruction in the sequence of statements that appears after the reserved word **begin** in their bodies. The common task is in a package which has the specification:

```
package Starting_gate is
  task T is
    entry gate (lowest_priority .. highest_priority - 1);
    entry start;
    pragma priority (highest_priority);
  end T;
end Starting_gate;
```

All compilation units that contain tasks request visibility to this package via a context clause. The main procedure calls the entry `Starting_Gate.T.start` which is accepted as the first executable instruction in the sequence of statements that appears in the body of task `Starting_Gate.T` (after the reserved word **begin**). This call to `Starting_Gate.T.start` in the main program must not occur before all pervasive tasks have begun execution; in the Ada systems we have tested, this can be ensured by giving the main procedure the lowest priority. The sequence of statements of `Starting_Gate.T` then consists of a series of loops. Each loop has a selective wait with an *else* option. The entry index for the accept statement

in a selective wait evaluates to a priority value and the loops are placed in decreasing order of priority, e.g.

```
task body T is
begin
  accept start;
  ...
  loop
    select accept gate (Priority_Value_1);
    else exit;
    end select;
  end loop;
  ...
end T;
```

Every other pervasive task calls entry gate(p) of Starting_Gate.T, where p is the priority of the calling task. Each loop accepts an entry call from a pervasive task in the program, and the sequence of loops and point of initiation of task Starting_Gate.T ensure synchronization in order of priority. An obvious limitation of this scheme is that the highest priority provided by the implementation cannot be used by a "client" task. Also, the priority range is implementation-dependent and so the package Starting_gate may need to be rewritten for different implementations. However, the need for modification is localized in this case, unlike in the *Delay* or the *Delay_Until* model, for example. Moreover, the priority range provided for an implementation is easily available, whilst the algorithm used by the compiler to determine elaboration order (one of the items needed to appropriately determine the start times in a task set implemented according to the *Delay* or the *Delay_Until* model) is not.

Pragma PERVASIVE

It would not be hard for the implementor of a compiler to introduce a pragma that allows the identification of pervasive tasks. Such tasks will then begin their initial execution as proposed in the preceding section for pervasive tasks. For many real-time systems it should be a simple matter to identify the tasks that require pragma PERVASIVE to preclude PIDE. For example, if RM scheduling with PCP is used, this pragma should appear in all client and monitor tasks that are enclosed

by library packages. Identification of all tasks requiring the pragma is certainly much simpler than, for example, the computations of the task start time which is needed in the *Delay* and *Delay_Until* models, or even the introduction of pragma ELABORATE to force the early elaboration of packages enclosing monitor tasks. A task containing pragma PERVASIVE should *not* be called from a package initialization part.¹³

Conclusions

Despite the success of RM scheduling and PCP, and the general correctness of the existing implementation template where semaphores are implemented using high priority monitor tasks, there remains the problem that task activation order may be able to destroy the proper functioning of the implemented model during the first execution cycle. The activation order is determined by the elaboration order, which differs between different Ada compilers and is somewhat hard to predict even in a single compiler. More consideration of task activation rules is appropriate by Ada 9X teams. Current application programs using the Ada-83 language may be rendered safer by applying the coding practices outlined above.

Acknowledgements

We thank IBM Owego for support of the State-Oriented System and Software Engineering Project, under which this paper was developed.

Bibliography

1. Ada Programming Language Reference Manual (LRM), ANSI/MIL-STD 1815 A, U.S. Government, Ada Joint Program Office, 1983.
2. Ada 9X Project Report, Draft Mapping Document, February 1991, Ada 9X Project Office.
3. Ada 9X Project Report, Revision Issues Release 1, April 1990, Ada 9X Project Office.
4. J.G.P. Barnes, *Programming in Ada*, 3rd Ed., Addison-Wesley, 1989.

5. M.W. Borger, M.H. Klein and R.A. Veltre, 'Real-time software engineering in Ada: Observations and guidelines,' Software Engineering Institute, Carnegie-Mellon University, Tech. Rep. No. CMU/SEI-89-TR-22, 1989.
 6. N.H. Cohen, *Ada as a second language*, McGraw-Hill, 1986.
 7. D. Cornhill, 'Session summary: tasking,' *Proc. 1st Int. Workshop on Real-Time Ada Issues, Ada Letters VII*, no. 6, pp. 29-32, 1987.
 8. E.W. Giering and T.P. Baker, 'Compile time scheduling of an Ada subset,' *Proc. 7th Washington Ada Symp.*, June 1990, pp. 143-55.
 9. J.B. Goodenough and L. Sha, 'The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks,' *Proc. 2nd Int. Workshop on Real-Time Ada Issues, Ada Letters VIII*, no. 7, pp. 20-31, 1988.
 10. L.C. Lander, S. Mitra, and T.F. Piatkowski, 'Priority inversion in Ada programs during elaboration,' *Proc. 7th Washington Ada Symp.*, June 1990, pp. 133-41.
 11. L.C. Lander, S. Mitra, and T.F. Piatkowski, 'Deterministic priority inversion in Ada selective waits,' *ACM Ada Letters*, vol. 10, no. 7, pp. 55-62, 1990.
 12. L.C. Lander, S. Mitra, N. Singhvi and T.F. Piatkowski, 'The elaboration order problem of Ada,' Dept. of Computer Science, SUNY-Binghamton, Tech. Rep. No. CS-TR-90-57, 1991.
 13. L.C. Lander and S. Mitra, 'Detection and avoidance of elaboration-time problems for multi-unit real-time Ada applications,' Dept. of Computer Science, SUNY-Binghamton, Tech. Rep. No. CS-TR-90-58, 1991.
 14. D. Locke, L. Sha, R. Rajkumar, J. Lehoczky and G. Burns, 'Priority inversion and its control: an experimental investigation,' *Proc. 2nd Int. Workshop on Real-Time Ada Issues, Ada Letters VIII*, no. 7, pp. 39-42, 1988.
 15. S. Mitra, 'Report on the μ SAN to Ada Mapping Algorithm,' Technical Report under Contract No. J40-6340F, Department of Computer Science, SUNY-Binghamton and IBM-Owego, 1991.
 16. T. F. Piatkowski, Lap-Kin Ip and Dayun He, 'State Architecture Notation and Simulation: A Formal technique for the Specification and Testing of Protocol Systems,' *Computer Networks* (6), pp. 397-418, 1982.
 17. L. Sha and J. B. Goodenough, 'Real-time scheduling theory and Ada,' *IEEE Computer*, vol. 23, no. 4, pp. 53-62, 1990.
 18. L. Sha, R. Rajkumar and J.P. Lehoczky, 'Priority inheritance protocols: An approach to real-time synchronization,' Computer Science Dept., Carnegie-Mellon University, Tech. Rep. No. CMU-CS-87-181, 1987.
 19. S. Vestal, 'On the accuracy of predicting rate-monotonic scheduling performance,' *Proc. Tri-Ada '90*, Dec. 1990, pp. 244-253.
 20. D.A. Watt, B.A. Wichmann and W. Findlay, *Ada Language and Methodology*, Prentice-Hall, 1987.
- Leslie C. Lander**—obtained his B.A. (1967) from Cambridge, U.K. and his Ph.D. (1973) from Liverpool, U.K. in Mathematics. He has taught in England, Germany, Perú and Venezuela. He has been at SUNY-Binghamton since 1984. His interests include programming languages, formal aspects of software engineering, and expert systems. He is a member of ACM and an associate member of IEEE.
- Sandeep Mitra**—graduated from the Indian Institute of Technology, Bombay, India, with a B.Tech (1985) and M.Tech. (1987) in Computer Science. From January 1988 until August 1991 he was a Research Assistant in the Department of Computer Science at SUNY-Binghamton, working for his Ph.D. in the area of formal specification techniques in software engineering for real-time systems. He is currently a Dissertation Year Fellow of the University. He is a student member of ACM and IEEE.
- Mailing address:
Department of Computer Science
The Thomas J. Watson School of Engineering,
Applied Science and Technology
State University of New York,
Binghamton, NY 13902-6000
- E-mail:
Bitnet: lander@bingvmb
Internet: lander@bingvaxu.cc.binghamton.edu,

Ada TASKING OPTIMIZATIONS

Arvind Goel
Unixpros Inc.
16 Birch Lane
Colts Neck, NJ 07722

and

Mary E. Bender
U.S. Army CECOM
Software Engineering Directorate
Ft. Monmouth, NJ 07703

ABSTRACT

This paper describes the results of implementing Ada tasking optimization techniques in an existing Ada Runtime System. These tasking optimizations include a) Habermann-Nassi's optimization of letting the client execute the rendezvous, b) Stevenson's optimization of letting the last task to arrive execute the rendezvous, and c) Frenkel's optimization of accept statements without bodies. Experiments were performed to determine the performance improvements resulting from these optimizations. It also draws conclusions and recommendations for Ada tasking optimization methods based on the result of the experiments.

1.0 INTRODUCTION

Real-time computer applications are characterized by interaction with real-world (physical) events, over whose timing they have little or no control. It is therefore a requirement of the software that it control its own execution timing so as to synchronize with the real world. If the timing of the software does not meet certain constraints, the entire system of which the software is a part may fail. Hence, for most real-time systems, the most important requirements are related to meeting timing constraints.

The performance of a real-time system can be affected adversely if the tasking implementation of a Ada compiler is inefficient. Introducing Ada tasks in a system incurs a certain amount of runtime overhead. This overhead includes task activation and termination, task scheduling and dispatching, context switching, propagation of exceptions, selection of an open entry in a selective wait statement, queuing management of entry calls, and allocation of task control blocks. Hence, efficient implementation of the Ada tasking mechanism is very important in order for real-time systems to meet their timing requirements.

2.0 Ada TASKING OPTIMIZATION ISSUES

Efficient implementation of the Ada tasking mechanism is very critical for real-time embedded systems. The semantics of Ada tasking as defined in the Ada Language Reference Manual (LRM) are complex and its use is typically expensive, both in terms of space and time. This expense can be reduced if the compiler recognizes frequently occurring idiomatic uses of tasking for which special purpose, less expensive implementations are possible. As described in [1], Ada rendezvous timing is the most important part of Ada tasking to consider when one is concerned with overhead, because it can happen many times during the execution of a program, as opposed to task creation, which occurs only a relatively small number of times. The primary overhead events of rendezvous are context switches and the execution of the select statement. A state switch is a change of control between two program units. It requires saving and restoring machine status and registers, parameter transmission, and the saving and formation of a referencing environment. State switches are required for procedure calls. A context switch is a state switch plus actions for transfer to a different thread of control. Typically, at least two to three context switches are required to perform rendezvous.

The major effort in this task is to implement optimizations that reduce the number of context switches required to perform a rendezvous and optimize the select/accept statements. Such optimizations include those proposed by Habermann-Nassi [2], Stevenson [3], Frenkel [4], and Shauer [5]. These proposals address the need for fewer context switches and/or optimization of the select/accept statement by proposing different techniques.

2.1 Ada Rendezvous

Multitasking in Ada consists of a set of inter-dependent language constructs built around the concept

of a rendezvous. An accept statement names an entry and associates the entry name with a body of code. Additionally, there can be list of formal parameters associated with the entry. Intertask communication is accomplished by "calling" an entry in another task. The task executing the accept statement and the task calling the entry meet in a rendezvous and stay locked in time until the body of code associated with the accept statement completes its execution, at which point both the client task and the server task continue asynchronously.

Based on the above description, two scenarios are possible. The client task can issue the call first, or the task owning the entry arrives at the accept for that entry first.

A normal rendezvous implementation works as follows:

a. Entry Call precedes accept.

A task calls the entry in the other task and blocks. When the server task executes an accept for this entry, it associates the actual parameters with the formals of the entry and executes the body textually associated with this accept. When the body completes, the client task is made ready to execute and the scheduler is entered to select a task to execute next. The server task continues to be available for scheduling.

b. Accept precedes entry call.

The task owning the entry arrives at the accept statement for the entry and blocks. A task issues an entry call and blocks. Then the task that is blocked on an accept starts executing, it executes the body, and continues as described in case 1.

Based on the above analysis, two context switches may be required if the task that made the entry call arrives first at the rendezvous, and three may be required if the task containing the accept statement precedes the entry call. At each scheduling point, there is the potential of a task context switch, in which the state of the task must be saved, and the context of another task restored and made ready to run.

2.2 Janus Ada Compiler System

The Janus Ada Runtime System has been used in this effort to implement several tasking optimizations, the majority of which attempt to reduce the number of context switches in order to speed up the rendezvous. The Janus Ada Compiler (Release 2.1.3) is self-hosted on a 386 computer (with 4 Mb RAM, 25 Mhz clock) running VENIX [2] Version 3.2.2, a real-time Unix by VenturCom Inc.

The Janus Ada RTS is written primarily in Ada with about 20 percent of the RTS written in Intel 80386 assembly language. The implementation of the tasking

mechanism is based on technical papers by T. P. Baker and G. P. Riccardi on Ada tasking [6], [7] [8].

In the Janus Ada RTS implementation, a Supervisor task is responsible for implementing Ada tasking semantics. It is important to understand the functions of the Supervisor, because most of the tasking optimizations implemented in this project involve modifications to the RTS Supervisor functionality. The Supervisor is implemented as a separate task that is created during the initialization of an Ada program. This task is not created if the Ada program does not use tasking constructs. The code generation phase of the Janus Ada compiler inserts calls to the tasking Supervisor when support for Ada tasking semantics is needed. The tasking Supervisor is responsible for the appropriate context switching and flag setting, and then it calls runtime routines to perform the needed tasking functionality.

2.3 Constraints During Implementation Of Tasking Optimizations

As stated earlier, the optimizations implemented in this effort are based exclusively on modification to the Ada Runtime System. The lack of access to source code of other parts of the Ada compiler (syntax and semantic analyzer, optimizer, and code generator) inhibited this project in the following ways:

- In some cases, it was impossible to recognize when a certain optimization could be applied since Pragmas or other mechanisms could not be used to indicate certain optimizations so that the code generator could produce code to implement them.

- Also, in certain cases, the code generator produced inline code that made it impossible to implement certain optimizations, e.g. the code generator allocates space for task stacks making it impossible to implement optimizations that remove a task completely.

3.0 HABERMANN-NASSI'S TASKING OPTIMIZATION

Habermann and Nassi in a paper published in 1980 [2], outlined two methods for reducing the speed of the rendezvous:

a. Letting the accept body run as part of the client's thread of control.

b. Conversion of a server to a monitor. Habermann-Nassi's second proposal is very similar to that of Shauer's proposal. Hence, its implementation is discussed along with Shauer's proposal in a later section.

The first proposal by Habermann-Nassi is an implementation method for simple rendezvous. The idea is simply to attempt to reduce the number of context

switches required for a rendezvous by letting the client task execute the accept statement. The client task executes the statements in the accept body, thus reducing the number of context switches by one. In the discussions that follow, a client task is one that makes the entry call, and the server task is one that accepts the entry call. It is assumed that the client task is of a higher priority than the server task.

There are two scenarios that have to be examined in order to determine the savings that result from this optimization.

a. **Client Arrives at the Rendezvous First.** If the client arrives at the rendezvous first, it is blocked. A context switch is made to the server, which then executes down to the accept statement. Then a context switch is made to the client, who executes the code in the rendezvous and continues execution. This optimization still results in two context switches and does not result in any savings.

b. **Server Arrives at the Rendezvous First.** If the server arrives at the rendezvous first, it is blocked and a context switch is made to the client. The client executes the rendezvous and continues execution. This rendezvous results in a single context switch, as opposed to three context switches if this optimization is not applied.

There is some additional overhead involved in letting the client execute the rendezvous. Semaphores are needed in order to ensure mutually exclusive access to the server task. Also, machine status and registers have to be saved during the execution of the rendezvous. Furthermore, the existing environment (scope) of the client is saved, and is temporarily replaced with that of the server. When the rendezvous is complete, the client's environment is reestablished. In essence, this implementation replaces two context switches, from the client to the server and back, with two state switches.

The next few sections describe the changes that were made to the Janus Ada Runtime System for implementing the Habermann-Nassi optimization of letting the client execute the rendezvous (server arrives first at the rendezvous). The case where the client arrives first at the rendezvous is not discussed as two context switches are still required, resulting in no savings in rendezvous time.

3.1 Modifications To Tasking Data Structures

Each task in an Ada program has a task control block (TCB) associated with it. The TCB contains information about the task with which it is associated. This information includes:

- the master of this task master's dependents
- state of the task (being activated, activated, already activated, passive, waiting for accept, complete,

terminated, callable, abnormal, need to raise exception)

- rendezvous status, status of various entries, entry queue status

- task priority, etc.

During the implementation of the Habermann-Nassi optimization, mutually exclusive access to the server task was needed. This was done to ensure that another task could not conceivably rendezvous with the server task, while the client is executing the body of the accept. To implement mutual exclusion, the TCB was modified to include the following variables:

a. **TCB_REC.mutex** - A semaphore used to control the critical sections which regulate rendezvous initiation and termination.

b. **TCB_REC.tasksem** - A semaphore used to control a task when it calls another task and has to wait for completion of the entry call.

To implement the semaphore mechanism, the wait and signal routines were implemented as part of the Supervisor code. The semaphore mechanism is implemented as an assembly language routine that uses the Intel 80386 bts (bit test and set) instruction for the wait mechanism and the btr (bit test and reset) instruction for clearing the bit once the process issuing the wait operation has completed. The initialization code for each task's TCB and any associated entry variables is generated when the task is initiated. The mutex semaphore is set to true and the tasksem semaphores are set to false, and each entry's waiting list is set to empty.

3.2 Implementation Of Habermann-Nassi's Optimization

In the explanation below, Task A's priority (client) is higher than that of task B (server). The case where the client (task A) arrives first at the rendezvous is not discussed as two context switches are still required, resulting in no savings in rendezvous time.

Case 1: Simple Rendezvous with No Select.

When task B arrives first at the accept, it is suspended as task A has not yet issued an entry call. Task A starts executing and when it reaches the statement B.X, an entry call is made. This results in a call to the Supervisor from task A. The call made is to perform the "entry call" functionality which requires a context switch. The tasking Supervisor first checks if the functionality being requested is to perform an "entry call". If this is true, then instead of first saving the context of task A, and then restoring its own context, the Supervisor checks if the corresponding server entry is open and if the server is waiting on an accept. If this is true, then no context switching is done, and the Supervisor calls the procedure to begin rendezvous. It also sets task B's tasksem

semaphore to true so that other tasks can not rendezvous with task B. In the Janus RTS implementation, the "begin rendezvous" function is also responsible for moving the entry call parameters from the client's stack to the server's stack. For the optimization, this step is not performed by the "begin rendezvous" function, which simply returns to the Supervisor.

The client task executes a portion of the server task, i.e. the accept body as part of its own thread of control. However, the body of the accept may require access to the state space local to the server task. Because of this, and because the language requires that exceptions raised during the rendezvous are propagated in both the client and server tasks, it is necessary to perform some administration before execution of the accept body.

The Supervisor code is modified to save the referencing environment of the client (task A). The information saved includes A's display registers, general purpose registers, and exception handling stack variables. Then the dynamic link (frame pointer) of the server task is obtained from its saved state. This is easily accessible, as the Supervisor has access to task B's task control block. The client task's dynamic link is stored in a reserved location in the stack frame. This frame is marked to indicate that, if an exception occurs, it must be propagated past this frame along both dynamic links. Then the server task's dynamic frame link is restored as the dynamic frame link of the client task. Other information restored includes task B's display registers, general purpose registers, and exception handling stack variables. This ensures that the exception handlers of B are in effect when the rendezvous is executed. After B's referencing environment is restored, transfer of execution is made to code inside the accept body.

During the rendezvous, reference to the entry call parameters is made via the stack pointer and any changes to B's variables are made on A's stack. After the code inside the accept statement has been executed, a call is made to the Supervisor to execute "end rendezvous" functionality. In the Janus implementation, the "end rendezvous" function is responsible for moving A's parameters from B's stack back to A. This code is not necessary as the rendezvous was performed on task A's stack and no parameters had been moved from task A's stack to task B's stack. Instead, the Supervisor code on return from "end rendezvous" function saves the registers of server task B, and restores client task A's registers as well as A's dynamic link. Then it checks to see if a higher priority task is ready to run, otherwise the execution of client task A is resumed. It also sets server task B's tasksem semaphore to false so that other tasks can now rendezvous with task B.

Case 2: Simple Rendezvous with Select.

The select statement implements multi-way waiting in Ada, with optional timeouts (the delay alternative) and a provision for proceeding if no entry call has been made (the else alternative). The transformation from single accept statements to select statements is straightforward as described below. Once again, task A's priority (client) is higher than that of task B (server).

Task B executes down to the select statement and it blocks. Task A starts executing and when it reaches the statement B.X, an entry call is made. The implementation is similar to that of case 1, until the point when B's referencing environment is restored. At this point, in case 1, transfer of execution is made to code inside the accept body. In the case of the select statement, since it is not known apriori which accept inside the select will be executed, task B is waiting to jump to the code inside the appropriate accept. This jump statement is executed by task A so that transfer is made to the correct accept statement. Hence, for the select statement, the difference is that the client also executes the jump statement which transfers control to the accept body of the appropriate entry.

Case 3: Delay Alternative in a Select.

In the examples above, the assumption is that there is no else or delay alternative. If there is a delay alternative in a select statement, then the implementation is the normal Janus implementation if no rendezvous occurs within the specified time. If a task does become ready to rendezvous during the specified time, then the implementation is the same as described in case 2.

Case 4: Else Alternative in a Select.

If there is an else alternative in the select statement, then the task containing the else alternative does not wait at all. If rendezvous is not possible, then it continues executing if it is the highest priority task that is ready to run. If rendezvous is possible, then implementation is similar to that of case 2.

3.3 RESULTS OF HABERMANN-NASSI'S OPTIMIZATION

The results obtained for the case when Habermann-Nassi optimization (letting the client execute the rendezvous) is applied are:

Description	Time (in microseconds)
Normal Rendezvous time (client arrives first)	1900
Normal Rendezvous time (client arrives last)	2100
Habermann-Nassi (client arrives last)	1789

From these results, the Habermann-Nassi optimization results in about 15 percent reduction in rendezvous timing. In Habermann-Nassi's optimization of letting the client execute the rendezvous, three context

switches are replaced by a single context switch and two state switches. Hence, it is reasonable to expect a 30 percent reduction in rendezvous timing as context switches are the most time consuming part of a rendezvous. The reduction in rendezvous timing is not higher due to the following reasons:

- Overhead due to semaphore implementation requiring mutually exclusive access to the server task.

- Overhead due to two context switches being replaced by two state switches. This involves saving the existing referencing environment (scope) of the client, replacing it with that of the server and then when the rendezvous is complete, restoring the client's environment. This implies that state switches for the Janus Ada RTS are quite expensive.

4.0 STEVENSON'S TASKING OPTIMIZATION

Stevenson proposed a method called "order of arrival" method, in which the last task to arrive at the rendezvous executes the rendezvous. The basis for the method is that the context switch at the beginning of a rendezvous can always be avoided by letting the last task to arrive execute the rendezvous. Stevenson does not specify whether control shifts to the client after the completion of the rendezvous, although there is little justification for that. It makes more sense to leave control in the executing task as long as possible.

The result of this approach is that each rendezvous requires only a single context switch (assuming equal priorities). Hence, this method saves one context switch per rendezvous, at the cost of an average of one state switch per rendezvous.

4.1 Modifications To Tasking Data Structures

During the implementation of the Stevenson optimization, mutually exclusive access to the server task was needed. This was done to ensure that another task could not conceivably rendezvous with the server task, while the client is executing the body of the accept. To implement mutual exclusion, the TCB was modified to include the following variables:

- a. TCB_REC.mutex - A semaphore used to control the critical sections which regulate rendezvous initiation and termination.

- b. TCB_REC.tasksem - A semaphore used to control a task when it calls another task and has to wait for completion of the entry call.

To implement the semaphore mechanism, the wait and signal routines were implemented as part of the Supervisor code. The initialization code for each task's TCB and any associated entry variables is generated when the task is initiated. The mutex semaphore is set to

true and the tasksem semaphores are set to false, and each entry's waiting list is set to empty.

4.2 Implementation Of Stevenson's Optimization

There are two scenarios: a) server arrives at the rendezvous first, and b) client arrives at the rendezvous first. When the server arrives at rendezvous first, this optimization is similar to that of Habermann-Nassi's optimization of letting the accept body run as part of the client's thread of control. The client task's priority is either equal or greater than that of the server. Implementation details are the same as in Habermann-Nassi.

When the client arrives at the rendezvous first (assuming that both client and server have equal priorities), it executes down to the entry call statement, and makes a call to the Supervisor to perform the "entry call" functionality.

When the client makes the entry call, the Supervisor knows the task on which the entry call has been made. Now instead of first saving the context of client task A, and then restoring the context of the Supervisor, the Supervisor checks if server B's entry is open and if the server is waiting on an accept. Since the server is not waiting on an accept, the context of task A is saved and the context of task B is restored. This results in saving of the time required to restore and then later on save the Supervisor's context, before task B's context is restored.

Then routines are called to transfer the parameters on the stack of task B, and the code inside the accept body is executed. After the rendezvous is complete, a call is made to perform the "end rendezvous" functionality. This does not cause a context switch and continues with the execution of the server (if no exception is raised). If an exception is raised, it is handled normally by the Janus Ada Runtime System.

4.3 Results of Stevenson's Optimization

The results obtained when Stevenson's optimization (last task to arrive execute the rendezvous) is applied are...

Description	Time (in microseconds)
Normal Rendezvous time (server arrives last)	1900
Stevenson Proposal (server arrives last and continues execution)	1605
Normal Rendezvous time (client arrives last)	2100
Stevenson's Optimization (client arrives last and continues execution)	1795

The case when the client arrives last is similar to the Habermann-Nassi optimization of letting the client execute the rendezvous. The case when the server arrives last and continues execution after executing the rendezvous has a 20 percent reduction in rendezvous

timing as compared to the normal Janus implementation. Most of the saving results from avoiding the time required to save and restore the Supervisor context when Supervisor calls are made during the rendezvous.

In Stevenson's order of arrival method, a single context switch is required when server arrives last at rendezvous. Hence, it is reasonable to expect a 40 percent reduction in rendezvous timing as context switches are the most time consuming part of a rendezvous. The reduction in rendezvous timing is not higher, because of the overhead due to semaphore implementation requiring mutually exclusive access to the server task.

5.0 FRENKEL'S TASKING OPTIMIZATION

Frenkel proposed an optimization that can be applied to accept statements without bodies. An accept statement that has no body does not require a state switch at the point of the rendezvous. The rendezvous, in this case, is merely a signal from one task to another, stating that execution has reached the point of interest (as marked by the entry call in the client and the accept statement in the server). Such accept statements require very little implementation overhead. If the client arrives first, it is blocked. When the server arrives, the client is moved to the ready queue. If the server arrives first, it is blocked. When the client arrives, the server must be moved to the ready queue.

This optimization is easy to implement, but difficult to detect. Due to lack of access to the code generator and other parts of the compiler, it cannot be detected easily if there are no statements inside the accept. A compiler dependent pragma could be used to indicate that there are no statements inside the accept. The code generator could use this information to avoid generating code to do anything except the blocking and unblocking of the first task to arrive at such an accept.

5.1 Implementation of Frenkel's Optimization

There are two scenarios: a) client arrives at the rendezvous first, and b) server arrives at the rendezvous first. In order to implement Frenkel's optimization, some of the modifications made to implement Habermann-Nassi and Stevenson optimizations have been reused. But, in order to understand the implementation of Frenkel's optimization, a complete description has been presented.

5.1.1 Client Arrives At the Rendezvous First

In the description below, Task A's priority (client) is higher than that of task B (server). Task B has an accept

statement with no code statements inside the accept body.

Task A executes down to the entry call statement B.X and makes a call to the Supervisor to perform "entry call" functionality. When the client makes the entry call, the Supervisor knows the task on which the entry call has been made. Now instead of first saving the context of task A, and then restoring the context of the Supervisor, the Supervisor checks if server B's entry is open and if the server is waiting on an accept. Since the server is not waiting on an accept, the context of task A is saved and the context of task B is restored. This results in saving of the time required to restore and then later on save the Supervisor's context, before task B's context is restored.

When task B executes the accept statement, the normal Janus Ada implementation calls the Supervisor to perform "selective wait" functionality, which checks to see if there is a task waiting on the entry queue for this entry. If there is a task waiting on this entry queue, then the functionality "begin rendezvous" is performed which is responsible for moving the parameters from the stack of the client to that of the server.

The "selective wait" functionality has been modified so that when it detects a task waiting on the entry queue, it simply returns a true value to the Supervisor. It also does not call the "begin rendezvous" routine so that parameters are not moved from the client to the server stack.

In the normal Janus implementation, after the code inside the accept statement has been executed, a call is made to the Supervisor to perform "end rendezvous" functionality, which is responsible for moving A's parameters from B's stack back to A. This code is not necessary as no actual rendezvous was performed and no parameters had been moved from task A's stack to task B's stack. Instead, the Supervisor code is modified so that the "end rendezvous" functionality does not call any of the Ada RTS routines.

Instead, the Supervisor code saves the registers of server task B, restores task A's registers, checks to see if a higher priority task is ready to run, otherwise it resumes the execution of client task A.

5.1.2. Server Arrives at the Rendezvous First

Task B executes down to the accept statement and is blocked waiting for the client to arrive. When the client makes the entry call, the Supervisor knows the task on which the entry call has been made. Now instead of first saving the context of task A, and then restoring the context of the Supervisor, the Supervisor checks if server B's entry is open and if the server is waiting on an accept. Since the server is waiting on an accept, task A is allowed to continue execution without restoring and

saving the Supervisor's context. This also results in saving of the time required to restore and then later on save the Supervisor's context.

When task A executes the entry call statement, the normal Janus Ada implementation calls the Supervisor to perform "entry call" functionality, which checks to see if the task to whom the entry call is made is ready for rendezvous. If there is a task ready, then the "begin rendezvous" functionality is performed which is responsible for moving the parameters from the stack of the client to that of the server. The "perform entry call" function has been modified so that when it detects a task ready for rendezvous, it simply returns a true value to the Supervisor. It also does not call the "begin rendezvous" function so that parameters are not moved from the client to the server stack.

In the normal Janus implementation, after the code inside the accept statement has been executed, a call is made to the Supervisor to perform "end rendezvous" function, which is responsible for moving A's parameters from B's stack back to A. This code is not necessary as no actual rendezvous was performed and no parameters had been moved from task A's stack to task B's stack. Instead, the Supervisor code is modified so that the "end rendezvous" function does not call any other Ada RTS routines.

Instead, the Supervisor code puts the server back in the ready queue, no context switches are made and the client continues execution.

5.2 Results of Frenkel's Optimization

The results obtained when Frenkel's optimization is applied are:

Description	Time (in microseconds)
Normal Rendezvous time (server arrives last)	1900
Frenkel's Optimization (server arrives last)	831
Normal Rendezvous time (client arrives last)	2100
Frenkel's Optimization (server arrives last)	831

This optimization results in the rendezvous timing reduced by more than 50 percent. But this optimization can only be applied when the servers have accept statements without bodies. If an Ada application is designed such that accept statements are only used for synchronization, then this optimization will result in at least 50 percent saving in rendezvous execution time.

6.0 HABERMANN-NASSI/SHAUER'S TASKING OPTIMIZATION

The type of compiler optimizations proposed by Habermann-Nassi and Shauer [5] are generally known as process removal optimizations. These are aimed at reducing the number of context switches by reducing the

number of processes that are actually executing in the final target code. Habermann-Nassi and Shauer have suggested that one way of performing process removal, in Ada, is to move, whenever possible, to a monitor-like representation of shared objects, for example, a shared object on which all operations must be mutually exclusive. The normal representation of this object would include the following synchronizing task:

```
task SYNCHRONIZER is
  entry OP1(...);
  entry OP2(...);
  ....
end SYNCHRONIZER;
```

```
task body SYNCHRONIZER is
begin
  loop
    select
      accept OP1(...) do
        ...
      end OP1;
    or
      accept OP2(...) do
        ...
      end OP2;
    or
      ...
    end select;
  end loop;
end SYNCHRONIZER;
```

Mutually exclusive behavior of these operators is guaranteed by the serial execution of accept statements by the single synchronizing task. A compiler that implements this optimization removes the synchronizing task altogether. Each accept statement is replaced by a procedure, and the client's entry calls are replaced by ordinary procedure calls. Mutual exclusion is effected through semaphore operations provided in the RTS; these operations are inserted before and after each procedure call, or at the start and finish of each procedure body. The effect of this transformation is to remove the context switches at the cost of performing primitive operations on a semaphore.

6.1 Monitors and Ada Tasks

Monitors were conceived by Brinch Hansen [9] and further developed by Hoare [10] as modules for managing mutually exclusive access to shared concurrently accessible resources. A monitor has a user interface that provides a set of procedures callable by users, a mechanism for sequential scheduling of calls by concurrently executing users, and an internal mechanism for suspending and subsequently reawakening processes

initiated by user calls. A monitor contains declarations of local data, a set of procedures callable by users of the monitor, and some initializing statements. At most one procedure of the monitor may be executing at any given time. When a user calls a procedure of the monitor, and there is no other concurrently executing procedure, the call can be executed immediately. If the monitor is already executing, the calling procedure is placed on a monitor queue from which called procedures are executed in the order of call.

Both monitors and Ada tasks provide the user with a set of callable resources (monitor procedures, task entries) that are serially reusable in the sense that only one procedure or task entry may be executing at a given time. However, the control mechanisms for scheduling monitor calls and task entries are very different. Completion of the monitor call returns the monitor to an "initial state" in which it may execute the next monitor procedure on its input queue. In contrast, completion of an accept statement in a called task is followed by execution of the statements following the accept statement until the task terminates or until another accept statement is encountered. Thus, an entry call can execute only if the task reaches a control point where the entry call is expected. One consequence of this difference is that waiting entry calls for a task are placed on different queues for each entry name while waiting procedure calls on a monitor are all placed on the same queue.

This difference in program structure reflects the fact that Ada tasks consist of sequentially executable statements while monitor procedures have no inherent sequential order. Sequential execution of monitor procedures must be realized by explicit scheduling using a nonlocal variable or semaphore, while sequential execution in Ada is inherent in the control structure. The monitor call mechanism is effectively a select statement with a common queue for all procedures. Operations of a monitor, like calls to subprograms contained in a package, cause the monitor to "wake up", to perform the required operation, and to go back to sleep.

6.2 Implementation of Habermann-Nassi/Shauer's Tasking Optimization

The transformation of a server to a monitor is generalized by Shauer as follows:

- a. Move all statements that are outside a rendezvous into the preceding rendezvous.
- b. Restructure each accept statement as a procedure.
- c. Encapsulate in a "start procedure" all statements that precede the first accept statement.
- d. Replace each guard by a test, such that a closed guard results in a suspension on a semaphore (or similar entity).

e. Replace each entry by the corresponding procedure call.

f. Replace the task initialization by a call to the start procedure.

To implement this optimization, the compiler can recognize when this optimization is possible and generate code accordingly. Failing this, a compiler-dependent pragma can be supported that allows the programmer to request that the associated task be replaced by a monitor. Removal of a task not only reduces the number of context switches required, but also reduces the general overhead of task management. Both these implementation options require changes to the code generator portion and other parts of the Ada compilation system. Due to lack of access to the source code for the syntax and semantic analyzer and code generator, it is impossible to recognize when this optimization is possible in order to implement it.

However, for comparison purposes and to determine the efficiency of this optimization, a monitor was developed in Ada for controlling access to data common to a group of processes (consumer-producer scenario). The data may be set by one or more processes or used by one or more processes. The monitor developed is used to control the reading and writing of data. Read and write operations are mutually exclusive and will not interfere with one another.

The monitor is written as a Ada package whose visible part contains subprogram declarations and some type declarations. The package `Read_Write_Manager` provides two procedures `Read` and `Write`. Two possible implementations are considered for `Read_Write_Manager`, the first using semaphores and the second using the Ada rendezvous mechanism.

a. This is the implementation that results after the server task has been converted to a monitor. The semaphore mechanism is implemented as an assembly language routine that uses the Intel 80386 `bts` (bit test and set) instruction for the wait mechanism and the `bt` (bit test and reset) instruction for clearing the bit once the process issuing the wait operation has completed.

b. This is the normal implementation when no optimization is performed. This implementation of `Read_Write_Manager` uses the Ada rendezvous mechanism. The procedures `read` and `write` inside the package `Read_Write_Manager` call entries `read` and `write` in an Ada task contained inside the same package.

6.3 Results of Habermann-Nassi/Shauer's Tasking Optimization

The results obtained when this optimization is applied are discussed below:

Case 1: One Consumer task and one producer task present in the system.

The time measured was for the consumer to read a data item after the producer has done a write. In the table below, the rendezvous read time is the time for a read operation using the rendezvous implementation, and the monitor read time is the time taken for a read operation using the monitor implementation.

Description	Time (in microseconds)
Rendezvous Read time	2050
Monitor Read Time	210

Case 2: Five consumers and five producers present in the system.

The time is measured to perform a read when a high degree of contention exists in the system - specifically while one task is writing data and there are two consumers and two producers queued on calls to the monitor. These tasks are blocked by the mutual exclusion mechanism. The timing measured are for a consumer that has issued a read and is queued after four read/write calls. In the rendezvous implementation scenario, the producer performs a rendezvous with the write entry, while two producer and two consumer tasks have issued entry calls and are waiting to rendezvous. The time measured is for a fifth task (consumer) that has issued a read call to get the data.

Description	Time (in microseconds)
Rendezvous Read time	8023
Monitor Read Time	5129

The results indicate that monitors are consistently more efficient than a server task implemented by a normal Ada method. But as case 2 indicates, the benefits of the monitor are reduced in the presence of a high degree of contention. When contention is present for both producer and consumer tasks, the time taken for a single read or write increases. This is because the monitor is blocking both clients sending and receiving data. For the normal rendezvous implementation, the time for a single read does not increase in proportion to the entry calls being made, as separate entry queues exist for each entry, and the select statement decides which accept statement to execute in a nondeterministic fashion. Hence, even though there are three consumers and two producers that are waiting on the entry queues, the third consumer could execute the read entry call as early as the third rendezvous or as late as the fifth rendezvous.

Another additional overhead of the monitor in situations of contention is that client tasks are blocked twice before the communication takes place, once at the mutual exclusion mechanism and once if the shared data structure is empty or full. On the other hand, there can be

only one block of an Ada task before the communication takes place. Hence, although monitors are very beneficial, this optimization is dependent on the application.

7.0 SUMMARY OF RESULTS, RECOMMENDATIONS AND CONCLUSIONS

Proposals for increasing the speed of Ada rendezvous have been tailored in a Ada Runtime System. These include a) Habermann-Nassi's optimization of letting the client execute the rendezvous, b) Stevenson's optimization of letting the last task to arrive execute the rendezvous, and c) Frenkel's optimization of accept statements without bodies. Experiments were performed to determine the performance improvements resulting from these optimizations.

Habermann-Nassi's optimization of letting the client execute the rendezvous resulted in a 15 percent reduction in rendezvous time over the normal implementation. The reduction in rendezvous timing was not more due to a) overhead of semaphore implementation requiring mutually exclusive access to the server task and b) overhead due to state switches - state switches for the Janus Ada RTS are quite expensive. Stevenson's optimization resulted in a 20 percent reduction in rendezvous time over the normal implementation. The reduction in rendezvous timing was not more due to overhead of semaphore implementation requiring mutually exclusive access to the server task. Frenkel's optimization resulted in a 50 percent improvement in rendezvous time, but this optimization can only be applied to accept statements without bodies.

The transformation resulting from Habermann-Nassi's and Shauer's optimization of converting a server to a monitor was programmed and its efficiency compared with that of a normal Ada implementation. The results indicate that conversion of a server to a monitor results in substantial reduction in rendezvous time when the contention in the system is low, but this time increases substantially when high contention is present in the system. Also, conversion of a server results in serialization of the software, as it forces all code to be executed as part of the monitor procedure call. Normally, the code after the accept statement is executed concurrently with the resumed client task. On uni-processor systems, this has no effect but this can result in performance degradation on a distributed system. The results of this effort show that tasking optimizations are beneficial and result in improved overall performance of a real-time application.

The overall conclusion is that current Ada tasking optimizations do improve Ada tasking performance. The significance of these improvements is application

dependent. For real-time embedded systems, even a 20 percent improvement in rendezvous time could be the difference in meeting their timing constraints. When serialization is undesirable, the recommendation for Ada compiler vendors is to apply, whenever possible, Habermann-Nassi's optimization (letting the client execute the rendezvous) and Stevenson's order-of-arrival method. For accept statements without accept bodies, Ada compiler vendors should implement Frenkel's optimization, as this will result in a substantial improvement in rendezvous time. Conversion of a server to a monitor is much harder, as it is extremely difficult to detect situations where it could be done. Compiler vendors could support a pragma that allows the programmer to request that the associated task be converted to a monitor. On uni-processor systems, conversion of a server to monitor results in substantial benefits in applications where contention is likely to be low.

REFERENCES

1. CECOM Software Engineering Directorate Technical Report C02 043NW 0001 00, "Ada Tasking Performance Issues", prepared by LabTek Corp., 23 February, 1990.
2. Habermann, A.N. et. al., "Efficient Implementation of Ada tasks", Carnegie-Mellon Univ., 1980.
3. Stevenson, D. R., "Algorithms for Translating Ada Multitasking", ACM SIGPLAN Symposium on Ada, Boston, Nov. 1980, pp. 166-175.
4. Frenkel, G., "Improving Ada Tasking Performance", Proc. ACM International, Workshop on Real-time Ada Issues, Ada Letters, Vol. 8, No. 7, 1987.
5. Shauer, J. Vereinfachung von prozess--Systemen durch sequentialisierung, 30/82, Institut for Informatik, Bericht.
6. T. P. Baker and G. A. Riccardi, "Ada Tasking: From Semantics to Efficient Implementation", IEEE Software, Volume 2, Number 2, March 1985,
7. G. A. Riccardi and T.P. Baker, "A Runtime Supervisor to Support Ada Task Activation, Execution and Termination", Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments, October, 1984, pp. 14-22.
8. G. P. Riccardi and T. P. Baker, "A Runtime Supervisor to Support Ada Tasking: Rendezvous and Delays," Proceedings of the Ada International Conference, 1985, pp. 329-342.
9. P. Brinch Hansen, Operating System Principles, Englewood Cliffs, NJ: Prentice-Hall, 1973.
10. C. A. R. Hoare, "Monitors: An operating system structuring concept", Communications of the ACM, 17, pp. 549-557, Oct. 1974, pp. 34-46.

ABOUT THE AUTHORS

Arvind Goel is the founder and president of Unixpros Inc., a firm specializing in research and development of Ada real-time systems, specifically in issues relating to Ada tasking, scheduling, Ada runtime system design using POSIX real-time extensions, etc. He has a MS degree in Computer Science from the University of Delaware.

Mrs. Mary E. Bender is a computer scientist with the Software Engineering Directorate, U.S. Army Communications-Electronics Command, Ft. Monmouth, NJ. She is the project leader for their technology program in Ada real-time applications and runtime environments. She received a B. A. degree in Computer Science from Rutgers University, New Brunswick, NJ.

Ada 9X PROJECT PANEL

Moderator: Chris Anderson, Elgin AFB

A REUSABLE ADA PACKAGE FOR SCIENTIFIC DIMENSIONAL INTEGRITY

George W. Macpherson
SoTech, Inc.
1330 Inverness Drive, Suite 315
Colorado Springs, CO 80910-3755

Abstract -- Our recent experience in operations Desert Shield and Desert Storm has clearly demonstrated the criticality of software reliability and that truly reliable software is indeed possible. One tool we can employ to enhance our software reliability is the classic scientific computational reliability technique of dimensional integrity, or units checking. The Ada programming language makes it easy to implement units checking. A PHYSICAL_QUANTITY type is declared which has entries for both the floating point numeric value of a variable, and integer entries for exponents of the fundamental physical units of mass, length, and time. The Ada feature of "operator overloading" allows creation of symbols to perform arithmetic and relational operations on these variables. For example, the multiplication operation "*" is overloaded so that the numeric values of the two operands are multiplied, and the exponents of their fundamental units are added. An ASSIGN function is then written to check that the units computed in an expression are equal to the units of the variable to which a new numeric value is assigned. Further reliability is added by "information hiding" through type limited private, and reusability and modifiability are added by making the package generic. For scientific software, this package is as reusable as package standard.

Index Terms -- Physical quantity, fundamental physical units, strong typing, operator overloading, information hiding, generic.

1. INTRODUCTION

Many techniques have been employed over the last 20 years to measure software quality and to predict the level of software reliability. These techniques include: formal specifications, mathematical proof of correctness, software and quality metrics, and mathematical models based on hardware reliability techniques. All of these techniques

have made contributions in some applications, but now the Ada language makes it very easy to incorporate into scientific software a classic technique which has been used to promote computational reliability for centuries.

2. BACKGROUND

In computation in the physical sciences, the convention is adopted that an algebraic symbol representing a physical quality, such as F, P, or V, stands for both a numeric value *and* a unit. For example, F might represent a force of 10 lbs., P a pressure of 14.7 lbs./in², and V a velocity of 32.2 ft/sec.

When we write the classic formula for motion under constant acceleration:

$$X = VT + 1/2 AT^2,$$

if X is in meters, then the terms VT and 1/2 AT² must also be in meters. Suppose T is in seconds. Then the units of V must be meters/sec and those of A must be meters/sec². (Of course, the factor 1/2 is a pure number, without units.) As a numeric example, let V = 10 meter/sec, A = 4 meters/sec², T = 10 sec. Then the above equation is written:

$$X = 10 \frac{\text{meters}}{\text{sec}} \times 10 \text{ sec} +$$

$$\frac{1}{2} \times 4 \frac{\text{meters}}{\text{sec}^2} \times 100 \text{ sec}^2$$

The units are now treated like algebraic symbols. The sec's cancel in the first term and the sec²'s in the second; thus

$$X = 100 \text{ meters} + 200 \text{ meters} = 300 \text{ meters}$$

Recall also that the units of all physical quantities can be expressed in terms of the fundamental units of Length, Mass, and Time, such as in our standard Meter-Kilogram-Second or MKS system.* If Force = Mass x Acceleration, then the units of force are Mass x Length/Time², and so on for torque, pressure, energy, etc.

3. ADA IMPLEMENTATION

To implement scientific dimensional integrity in Ada, a first and natural approach would be to use derived types and employ the Ada strong compile time type checking. In the example above, we could declare derived types for length, velocity, acceleration, and time. But in developing a general technique applicable to all scientific programming, a severe problem soon becomes apparent: We would need a separate derived type declaration for every conceivable kind of unit, such as pressure, torque, momentum, etc. The even more severe problem of derived types for intermediate results becomes apparent from the very simple example equation above. The equation contains the factor T², time-squared. If we have a derived type for time, we must also have a derived type for time-squared. Further, to compute the term AT², we must overload the multiplication operator to accept operands of type acceleration and time-squared, and return a value of type length. Thus, for even simple equations, the use of derived types becomes burdensome; for complex equations, it becomes impractical. Gehani⁵ addresses at length the issue of "Units of measure versus derived types" and concludes that the units approach is better.

A much simpler technique is to develop an abstraction of all physical quantity types and emulate the method we use for units checking in manual calculations, that is, express the problem only in terms of the fundamental units of Length, Mass, and Time.

* Of course, if we add the fundamental unit Q for charge, we can express all electrical units such as volts, Ohms, Webers, etc., in the MKSQ system. For simplicity, this paper deals only with Mass, Length, and Time units.

From the example equation above, we could write

$$V = 10 \text{ meters/sec} = 10.0 \text{ meters}^1 \text{ sec}^{-1}$$

$$A = 4 \text{ meters/sec}^2 = 4.0 \text{ meters}^1 \text{ sec}^{-2}$$

$$T = 10 \text{ sec} = 10.0 \text{ sec}^1$$

$$X = \text{meters}^1$$

To represent a physical quantity in software, we can create a record which contains a floating point entry to represent the numeric value, and three integer entries which represent the unit of the physical quantity in terms of exponents of the fundamental units of Length, Mass, and Time. The scheme is illustrated in Figure 1.

	NUMERIC VALUE	EXPONENTS OF UNITS		
		Length	Mass	Time
V	10.0	1	0	-1
A	4.0	1	0	-2
T	10.0	0	0	1
X		1	0	0

Figure 1. Computer Representation of Physical Quantities

Certain features of the Ada language make this scheme very easy to implement. These features are:

- a. Strong Typing
- b. Record Descriptions
- c. Operator Overloading

The features of strong typing and record descriptions allow us to create a template for

physical quantities. To implement this in Ada, we should first realize there are many systems of fundamental units (Meter, Kilogram, Second, ... Centimeter, Gram, Second, ... Foot, Slug, Second, ...), and further that the concept of dimensions extends beyond physics problems (a subject addressed later in this paper). To form a record template for creating items of type PHYSICAL_QUANTITY, we need to know the system of units the user wishes to use and the numeric precision to be employed. We can do this through a package specification as follows:

```
generic
  type Units_Type is (< >);
  type Numeric_Value is digits < >;
package Units_Integrity is
  Units_Exception:exception;
  type Unit_Array is array
    (Units_Type) of Integer;
  Z:Unit_Array;
  type PHYSICAL_QUANTITY is limited
    private;
function Init
  (Numerical_Value:Numeric_Value;
   Units:Unit_Array)
  return PHYSICAL_QUANTITY;
:
private
  type PHYSICAL_QUANTITY is
    record
      Value:Numeric_Value;
      Unit:Unit_Array;
    end record;
end Units_Integrity;
```

The reason for making PHYSICAL_QUANTITY limited private is to enforce use of an Assign procedure. This issue will be discussed later.

Now having a type definition and an initialization function, we can instantiate the Units_Integrity package, and create and initialize objects for our equation of distance under uniform acceleration:

```
with Units_Integrity;
procedure Example is
  type MKS_Units is (Meters, Kilograms,
    Seconds);
```

```
package Units_Pack is new
  Units_Integrity
    (Units_Type => MKS_Units,
     Numeric_Value => Float);
use Units_Pack;
X:PHYSICAL_QUANTITY:=
  Init (Numerical_Value => 0.0
        Unit => (Meters => 1,
                  Kilograms => 0,
                  Seconds => 0));
V:PHYSICAL_QUANTITY:=
  Init (Numerical_Value => 10.0,
        Unit => (Meters => 1,
                  Kilograms => 0,
                  Seconds => -1));
:
end Example;
```

and so on for A and T.

Of course we need to provide the user with a means to compute with these objects. The Ada feature of "operator overloading" allows us to give new meanings to the arithmetic operators, +, -, *, /, **, in order to perform arithmetic operations on these variables of type PHYSICAL_QUANTITY.

For example, the multiplication operator "*" may be overloaded to operate on PHYSICAL_QUANTITY types. In multiplication of PHYSICAL_QUANTITYs, we wish to multiply the numeric values and add the exponents of dimensions. To do this, we can write:

```
function "*" (X,Y: PHYSICAL_QUANTITY)
  return PHYSICAL_QUANTITY is
begin
  for I in Unit_Array ' range loop
    Z (I):=X.Unit(I)+Y.Unit(I);
  end loop;
  return (X.Value*Y.Value, Z);
end "*";
```

In this example, the operator "*" is extended to perform the operations we require on variables of type PHYSICAL_QUANTITY.

To incorporate an addition operator for PHYSICAL_QUANTITY we can write:

```
function "+" (X, Y: PHYSICAL_QUANTITY)
  return PHYSICAL_QUANTITY is
begin
  if X.Units /= Y.Units then
    raise Units_Exception;
  else
    return (X.Value + Y.Value,
            X.Unit);
  end if;
end "+";
```

In this case, we first check the units of the operands to insure that we are not adding apples to oranges, then add the numeric values of the operands. If the units of the two operands are not equal, "+" raises the exception Units_Exception, and further operation is determined by a Units_Exception exception handler of the users choice.

In Ada, we cannot overload the assign operator ":", but we can write a very simple procedure which will provide assignment and do what we really want to do - check for dimensional integrity. The Ada code for this is:

```
procedure Assign
  (Source: out PHYSICAL_QUANTITY;
   Target: in out PHYSICAL_QUANTITY) is
begin
  if Source.Unit/=Target.Unit then
    raise Units_Exception;
  else
    Target:=Source;
  end if;
end Assign;
```

To go back to our original equation example, we can write:

```
Assign (Source => V*T + 0.5 *A*T**2,
        Target => X);
```

which will operate correctly. However, if we had made a coding error and used an asterisk instead of a plus sign in the addition of the two terms, we would have:

```
Assign (Source => V*T * 0.5 *A*T**2,
        Target => X);
```

which would raise Units_Exception.

As noted above, type PHYSICAL_QUANTITY is limited private. Had we made the type unlimited or even limited, the assign operator ":" would be available to the user with the risk that assignment could take place without units being checked.

The complete package specification and body of Units_Integrity is listed in Appendix A. A sample program for computing motion under constant acceleration is shown in Appendix B.

4. "BILITIES"

In creating software, we must address certain "Bilities." Among these are:

- a. Reliability
- b. Portability
- c. Reusability
- d. Modifiability

Concerning reliability in this proposed package, we will claim that the package itself is very short, very simple, and consists of very stereotyped code; therefore, should be quite reliable. Concerning portability, the package is written in full compliance with ANSI/MIL-STD-1815-A, and contains no implementation-dependent features and, therefore, should be portable.

Concerning reusability, the package is applicable to any effort addressed to scientific computation. A further reusability feature of the Dimensional Integrity Package is that it is generic and the user has the option to specify the precision of the floating point value of a PHYSICAL_QUANTITY variable, and to specify the fundamental units to be monitored, e.g., Length, Mass, and Time, or Length, Mass, Time, and Charge. But these are Ada implementation details which are covered in the next section.

Concerning modifiability, it is clear that there is one major modification we may wish to make once we have this package incorporated into an application. The Package for Dimensional Integrity requires more storage and more CPU time than existing methods, since units exponents are tracked and manipulated. We read often in the literature today that since hardware is developing so rapidly, we don't

have to worry so much about storage space and CPU time. However, it has been the experience of this author that in real-time applications, storage and CPU resources are still a major concern, and probably will be for some time to come. Therefore, the major modification we may wish to make is to drop the units checking altogether. Once we have executed every line of code in our program, all units checks have been performed and we may very well want to turn off units checking.

If computational speed and memory requirements dictate, applications software developed with the Scientific Dimensional Integrity Package can easily be reverted to performing only the numerical operations, with units checking eliminated. This can be done by compiling the applications software with a different package. This new package would contain the declaration:

```
Subtype PHYSICAL_QUANTITY is
Numeric_Type;
```

The arithmetic and relational operators would not be overloaded, and the Assign procedure in the new package would not perform units checking. Also, the Init function would not deal with units exponents. The applications software developed with the Dimensional Integrity Package can be reverted to a non-dimensional checking mode by making no changes other than to reference the new package. Such a package, "Units_Integrity_Retro", is listed in Appendix C.

5. IMPLEMENTATION DETAILS

Since Ada is very powerful, it is necessarily somewhat complicated. While the sections above present the major rational and implementation concepts of the Dimensional Integrity Package, certain implementation details need to be addressed.

5.1 Generics

The version of the Dimensional Integrity Package listed in Appendix A has been made generic to provide for a user-defined system of significant digits, and a user-defined system of fundamental units. We mentioned earlier that if we included a fundamental unit for charge, we could express electrical units. If we included a

fundamental unit for temperature, and a fundamental unit for light intensity, we could express the units of thermodynamics and optics. The user of this package can select the fundamental units through the generic parameter, Units_Type, in line 3, and can select precision through the generic parameter, type Numeric_Value, in line 4. In our example (Appendix B), we have instantiated the package with six significant digits and the fundamental units of Slugs, Feet, Seconds on line 3-6 of the applications program. It would be possible to specify an absolute precision rather than a relative precision by using the reserved word, "delta", instead of "digits" on line 4, in the Dimensional Integrity Package.

5.2 Information Hiding

One feature of the Ada language that greatly promotes reliability is "information hiding." That is, the structure of data representation is kept private within a portion of a utility package and variables of the private type may be manipulated by the applications program only by using the functions and procedures provided in the utility package specification. Thus, in our example on lines 10-11 of the applications program, when the variable X has been created with the dimension of length, the units of X can not be changed by subsequent statements. Only a new call on the function Init, with X as a receiver, can give new values to the units of X.

In this version of our package, we have made PHYSICAL_QUANTITY a limited private type (line 8). This enforces use of the Assign procedure to assign new values to the numeric part of X, and thus, units checking is performed each time the variable is assigned a new numeric value. If PHYSICAL_QUANTITY were only private instead of limited private, the standard operations of assignment, equality, and inequality would be available to the applications program, and units integrity could be compromised.

5.3 Multiple Overloading

In Section 3 we showed an example of overloading the "*" operator to multiply two variables of type PHYSICAL_QUANTITY, but, of course, we need to multiply a floating point times a PHYSICAL_QUANTITY, and to be user-

friendly, we may as well allow the reverse order, a PHYSICAL_QUANTITY times a floating point. Thus, three overloads of "*" are required, the specifications for which are shown on lines 14 through 18. Similar considerations apply to the division operator "/" but for addition, subtraction, and the relational operators, this version of the package requires that both operands be of type PHYSICAL_QUANTITY. Also, since units checking is performed repeatedly, the procedure Mix-Check has been included in the package body.

6. OTHER APPLICATIONS

As noted above, the concept of dimensional analysis extends beyond problems in physics. In the NORAD Tactical Decision Aid (TDA) program there is a need to track the amount of fuel remaining in an interceptor, given parameters such as initial fuel, fuel consumption rate, interceptor speed, distance traveled, and time. A simplified overview of the way the Units_Integrity is employed in TDA is shown in Appendix D.

The parameters of the problem are set by the Init function, the only means of establishing unit dimensions without checking. Time_Flowed and the Fuel-Remaining are computed under the units scrutiny of the arithmetic operators and the Assign procedure.

The Units_Integrity concept can be applied to any situation involving clearly definable units. To apply the package to a payroll problem, the user could instantiate the package with:

```
type Payroll_Units is (Employees,  
Dollars, Hours);
```

To address software estimates, the package could be instantiated with:

```
type Software_Units is (Dollars,  
Lines_of_Code, Days);
```

7. OTHER APPROACHES

The concept of employing dimensional analysis in software was proposed long before the advent of the Ada language. The earliest reference to the subject this author has read is the 1977 Gehani paper.¹ In this paper he

states, "In everyday life as well as in science, engineering, banking, sampling inspection, ... , etc., the importance of specifying the units of the quantity or quantities is universally recognized ..." In a later paper, Gehani⁵ discusses the merits of using derived types versus units of measure, concluding that the units of measure approach is better. House³ gives a critique of Gehani's first paper and proposes a compiler extension which would perform a static check of units at compile time. He also addresses the problem of fractional powers of fundamental units, such as one would get when taking the square root of speed. Karr and Loveman⁴ take a linear algebra approach to the problem and address the problem of unit conversions (feet, inches, meters, ...) with the concept of "commensurate units." The most recent paper the author has read on the subject is by Hilfinger⁹. He proposes representing a physical quantity by a record type having a single element to hold the numeric value, and integer discriminates to hold the values of exponents of basic units. Hilfinger quotes a remark by O'Keefe⁸ which prompted him to publish his paper:

I am astounded that the longstanding idea [physical dimensions as types] did not make it into Ada [SIC], so much of less potential benefit having been included.

8. COST/BENEFIT

It is certainly clear that the process of units checking will increase costs at development time and we must project whether or not the benefits would justify the investment. An actual Cost/Benefit analysis will be possible only after data has been collected on projects using the concept, but here are three items we can keep in mind in making our projection:

8.1 Hardware Precedents

To gain insight on why hardware development has so greatly out-stripped software development, the author has attended presentations by the hardware people. It seems they are forever in a real estate battle. When you have a huge amount of functionality that has to go on a one square inch chip, territorial

concerns become prime. Yet in the final result, it appears to me that they (at least today's survivors) have found it worthwhile to invest the real estate, time, and money to include self-checking in the chip. Their decision is more whether to include fault isolation along with fault detection, rather than "is fault checking worthwhile."

8.2 Traditional Engineering

True today, as it has been long before computers arrived, engineers perform a dimensional analysis of the equations of any new development project. The process is time consuming and tedious, but long experience has proved that it is extremely cost-effective.

8.3 Programming Language Philosophy The philosophy of programming languages has changed. Back in FORTRAN days, the thrust was to make it as quick and easy as possible for the programmer to get his program running. We didn't even have to declare out variables - the compiler figured that out from context. But, we discovered that maintenance, not initial development, was the major source of cost. Now, in Ada, we not only have to declare our variables, but also declare their type, give their binding mode when used as subprogram parameters, and perform other time-consuming and expensive chores at development time. I submit this cost has been justified and that we can reasonably expect that incorporating the basic, fundamental engineering principle of dimensional analysis will also be justified.

9. CONCLUSION

Like O'Keefe, this author is astounded that the basic and fundamental engineering principle of dimensional analysis has not become a cardinal rule in software applications. There are several ways to go about it, and none are as tedious as the manual methods routinely and religiously employed by physicists and engineers. Compiler modification which would perform units verification at compile time would certainly be nice, although I'm not holding my breath until this is incorporated in ANSI/MIL-STD-1815, but great enhancement of software reliability is possible with existing compilers. The author has found the package

presented here adequate for his own purposes, and believes two ideas are present here which have not been published before:

1. Conveying the kind and number of units employed through a generic enumeration type.
2. Turning off units checking in a validated program with a retrograde package.

The software engineer who wishes to employ true engineering in his/her code will select methods which best fit the application, and in vigorously exercising these basic principles, will no doubt discover techniques which would not be found by academic speculation.

10. ACKNOWLEDGEMENTS

I wish to thank Michael D. Colgate of Loral Command and Control Systems, who originated many of the ideas expressed here, and the ANCOAT Technical Program Committee, who made helpful recommendations. I also wish to thank Darren Stautz, Kirstan Vandersluis, and Mike Winterbottom of SofTech, and Gordon Girod of Pentastar Support Services, Inc., for review and suggestions, and I thank Marti Devine of SofTech for technical preparation of this paper.

11. REFERENCES

1. Gehani, N. "Units of Measure as a Data Attribute." *Comput. Lang.* 2, 3 (1977), 93-111.
2. Hilfinger, P.N. "Abstraction Mechanisms and Language Design." MIT Press, Cambridge, Mass., 1983.
3. House, R.T. "A Proposal for an Extended Form of Type Checking of Expressions." *Comput J.* 26, 4 (Nov 1983), 366-374.
4. Karr, M., and Loveman, D.B. III "Incorporation of Units into Programming Languages." *Commun. ACM* 21, 5 (May 1978) 385-391.
5. Gehani, N. "Ada's Derived Types and Units of Measure." *Softw. Pract. Exper.* 15, 6 (Jun 1985), 555,569.
6. O'Keefe, R.A. "Alternatives to Keyword Parameters." *ACM SIGPLAN Not.* 20, 6 (Jun 1985), 26-32.
7. Zorn, B.G. "Experience with Ada Code Generation." Tech. Rep. UCB/CSD 85/249, Computer Science Division, Univ. of California, Berkeley, Jun 1985.
8. O'Keefe, R.A. "Alternatives to Keyword Parameters." *ACM SIGPLAN Not.* 20, 6 (Jun 1985), 26-32.
9. Hilfinger, P.N. "An Ada Package for Dimensional Analysis." *ACM Transactions on Programming Languages*, Vol 10, No. 2, (Apr 1988).

The Author:

George W. Macpherson received a B.S. degree in Electrical Engineering from the USAF Institute of Technology, Wright-Patterson AFB, Ohio, and M.S. degrees in Electrical Engineering and Mathematics from the University of Michigan, Ann Arbor. He served four years on the Computer Science faculty of the Air Force Academy, where he wrote a textbook on the ALGOL programming language. He has extensive experience in C³ and scientific software, and has originated and taught courses in the Ada language. He presently serves as Chief Scientist for SofTech Colorado Springs Operations, is responsible for Ada training at the Colorado Springs facility, and supports the NORAD Granite Sentry Software Tools Group. Readers may write to G.W. Macpherson at SofTech, Inc., 1330 Inverness Drive, Suite 315, Colorado Springs, CO 80910.

APPENDIX A (1). UNITS_INTEGRITY PACKAGE SPECIFICATION

```

1  generic
2    -- An Ada package for Scientific Dimensional Integrity.
3    type Units_Type is (<>);      -- User specifies the system of units.
4    type Numeric_Value is digits <>; -- User specifies significant digits.
5  package Units_Integrity is
6    Units_Exception:exception; -- Raised when units integrity is violated.
7    type Unit_Array is array (Units_Type) of integer;
8    type Physical_Quantity is limited private; -- Enforces use of 'Assign'.
9
10   -- Arithmetic operators:
11   function "+" (X,Y:Physical_Quantity) return Physical_Quantity;
12   function "-" (X,Y:Physical_Quantity) return Physical_Quantity;
13   function "-" (X :Physical_Quantity) return Physical_Quantity;
14   function "*" (X,Y:Physical_Quantity) return Physical_Quantity;
15   function "*" (X:Physical_Quantity;
16                 Y:Numeric_Value) return Physical_Quantity;
17   function "*" (X:Numeric_Value;
18                 Y:Physical_Quantity) return Physical_Quantity;
19   function "/" (X,Y:Physical_Quantity) return Physical_Quantity;
20   function "/" (X:Physical_Quantity;
21                 Y:Numeric_Value) return Physical_Quantity;
22   function "/" (X:Numeric_Value;
23                 Y:Physical_Quantity) return Physical_Quantity;
24   function "***" (X:Physical_Quantity;
25                  Y:Positive) return Physical_Quantity;
26
27   -- Relational operators:
28   function ">" (X,Y:Physical_Quantity) return boolean;
29   function "<" (X,Y:Physical_Quantity) return boolean;
30   function ">=" (X,Y:Physical_Quantity) return boolean;
31   function "<=" (X,Y:Physical_Quantity) return boolean;
32   function "=" (X,Y:Physical_Quantity) return boolean;
33
34   -- To initialize numeric and dimensional values:
35   function Init(Numeric_Value:in Numeric_Value;
36                Units :in Unit_Array) return Physical_Quantity;
37
38   -- To perform the function of the assign operator ":=":
39   procedure Assign(Source:in Physical_Quantity;
40                   Target:in out Physical_Quantity);
41   -- To make Numeric_Value visible:
42   function Numeric_Part (Object:Physical_Quantity) return Numeric_Value;
43
44   -- To Make Units visible:
45   function Units_Part (Object:Physical_Quantity) return Unit_Array;
46 private
47   type Physical_Quantity is
48     record
49       Value:Numeric_Value := 0.0;
50       Unit :Unit_Array := (Unit_Array'range => 0);
51     end record;
52 end Units_Integrity;
```

APPENDIX A (2). UNITS_INTEGRITY PACKAGE BODY

```

1  package body Units_Integrity is
2      Z:Unit_Array;
3      procedure Mix_Check(X,Y:in Physical_Quantity)is
4      begin
5          if X.Unit/=Y.Unit then
6              raise Units_Exception;
7          end if;
8      end Mix_Check;
9      function "+"(X,Y:Physical_Quantity)return Physical_Quantity is
10     begin
11         Mix_Check(X,Y);
12         return (X.Value+Y.Value, X.Unit);
13     end "+";
14     function "-"(X,Y:Physical_Quantity)return Physical_Quantity is
15     begin
16         return (-X.Value,X.Unit);
17     end "-";
18     function "--"(X :Physical_Quantity)return Physical_Quantity is
19     begin
20         return (-X.Value, X.Unit);
21     end "--";
22     function "*" (X,Y:Physical_Quantity)return Physical_Quantity is
23     begin
24         for I in Unit_Array'range loop
25             Z(I):=X.Unit(I)+Y.Unit(I);
26         end loop;
27         return (X.Value*Y.Value,Z);
28     end "*";
29     function "*" (X:Physical_Quantity;Y:Numeric_value)
30                                     return Physical_Quantity is
31     begin
32         return (X.Value*Y,X.Unit);
33     end "*";
34     function "*" (X:Numeric_value;Y:Physical_Quantity)
35                                     return Physical_Quantity is
36     begin
37         return (X*Y.Value,Y.Unit);
38     end "*";
39     function "/"(X,Y:Physical_Quantity)    return Physical_Quantity is
40     begin
41         for I in Unit_Array'range loop
42             Z(I):=X.Unit(I)-Y.Unit(I);
43         end loop;
44         return (X.Value*Y.Value,Z);
45     end "/";
46     function "/"(X:Physical_Quantity;Y:Numeric_value)
47                                     return Physical_Quantity is
48     begin
49         return (X.Value/Y,X.Unit);
50     end "/";
51     function "/"(X:Numeric_value;Y:Physical_Quantity)
52                                     return Physical_Quantity is
53     begin
54         for I in Unit_Array'range loop
55             Z(I):=-Y.Unit(I);
56         end loop;
57         return (X/Y.Value,Z);
58     end "/";

```

APPENDIX A (3). UNITS_INTEGRITY PACKAGE BODY (CONTINUED)

```

59  function "***"(X:Physical_Quantity;Y:Positive)
60                                     return Physical_Quantity is
61  begin
62      for I in Unit_Array'range loop
63          Z(I):=X.Unit(I)*Y;
64      end loop;
65      return (X.Value**Y,Z);
66  end "***";
67  function ">" (X,Y:Physical_Quantity) return boolean is
68  begin
69      Mix_Check(X,Y);
70      return X.Value>Y.Value;
71  end ">";
72  function "<" (X,Y:Physical_Quantity) return boolean is
73  begin
74      Mix_Check(X,Y);
75      return X.Value<Y.Value;
76  end "<";
77  function ">=" (X,Y:Physical_Quantity) return boolean is
78  begin
79      Mix_Check(X,Y);
80      return X.Value>=Y.Value;
81  end ">=";
82  function "<=" (X,Y:Physical_Quantity) return boolean is
83  begin
84      Mix_Check(X,Y);
85      return X.Value<=Y.Value;
86  end "<=";
87  function "=" (X,Y:Physical_Quantity) return boolean is
88  begin
89      Mix_Check(X,Y);
90      return X.Value=Y.Value;
91  end "=";
92  function Init( Numerical_Value:in Numeric_Value;
93                Units           : in Unit_Array)
94                                     return Physical_Quantity is
95  begin
96      return (Value => Numerical_Value,
97             Unit  => Units);
98  end Init;
99  procedure Assign(Source:in      Physical_Quantity;
100                 Target:in out Physical_Quantity) is
101      -- "in out" for Mix_Check
102  begin
103      Mix_Check(Target,Source);
104      Target:=Source;
105  end Assign;
106  function Numeric_Part (Object:Physical_Quantity)
107                                     return Numeric_Value is
108  begin
109      return Object.Value;
110  end Numeric_Part;
111  function Units_Part   (Object:Physical_Quantity)
112                                     return Unit_Array is
113  begin
114      return Object.Unit;
115  end Units_Part;
116
117  end Units_Integrity;

```


APPENDIX B. EXAMPLE PROGRAM

```

1  with Units_Integrity;
2  procedure Example is
3  -- Compute  $X = V \cdot T + 1/2 \cdot A \cdot T^2$ 
4    type Units_Type is ( Slugs, Feet, Seconds );
5    package Units_Pkg is new Units_Integrity(Units_Type =>Units_Type,
6                                              Numeric_Value => Float);
7    use Units_Pkg;
8    X,V,A,T: Physical_Quantity;
9  begin
10   X := Init(Numerical_Value => 0.0, -- Named Notation
11            Units           => (Slugs => 0, Feet => 1, Seconds => 0));
12   V := Init(Numerical_Value => 4.0, -- Named Notation
13            Units           => (Slugs => 0, Feet => 1, Seconds => -1));
14   A := Init(4.0, (0,1,-2 ));      -- Positional Notation
15   T := Init(10.0, (0,0,1 ));      -- Positional Notation
16   Assign( Source => V*T+0.5*A*T**2, -- O.K.
17          Target => X);
18   -- Assign(Source => V*T+0.5*A*T*2,      -- will not compile
19   -- .....^..
20   --      Target => X);
21   Assign( Source => V*T*0.5*A*T**2, -- raises exception
22   -- .....^.....
23   --      Target => X );
24 end Example;

```

APPENDIX C. A UNITS_INTEGRITY RETROGRADE PACKAGE

```

1  -- To turn off units checking after verification with package
2  -- Units_Integrity, the applications software is recompiled
3  -- with the following package:
4  generic
5      type Units_Type is (<>);
6      type Numeric_Value is digits <>;
7  package Units_Integrity_Retro is
8      Units_Exception:exception;
9      type Unit_Array is Array (Units_Type) of Integer;
10     subtype Physical_Quantity is Numeric_Value;
11     procedure Assign(Target:in out Physical_Quantity;
12                     Source:in    Physical_Quantity);
13     function Init ( Numerical_Value:in Numeric_Value;
14                     Units           :in Unit_Array)
15                     return Physical_Quantity;
16     function Numeric_Part (Object:Physical_Quantity) return Numeric_Value;
17     function Units_Part   (Object:Physical_Quantity) return Unit_Array;
18 end Units_Integrity_Retro;
19
20 package body Units_Integrity_Retro is
21     procedure Assign(Target:in out Physical_Quantity;
22                     Source:in Physical_Quantity) is
23     begin
24         Target:= Source;
25     end Assign;
26     function Init (Numerical_Value:in Numeric_Value;
27                     Units           :in Unit_Array)
28                     return Physical_Quantity is
29     begin
30         return Numeric_Value;
31     end Init;
32     function Numeric_Part (Object:Physical_Quantity)
33                             return Numeric_Value is
34     begin
35         return Object;
36     end Numeric_Part;
37     function Units_Part   (Object:Physical_Quantity)
38                             return Unit_Array is
39     begin
40         Null_Array: Unit_Array := (others => 999);
41         return Null_Array;
42     end Units_Part;
43 end Units_Integrity_Retro;

```

APPENDIX D. FUEL CALCULATION WITH UNITS_INTEGRITY

```

with Units_Integrity;
package body TDA_Overview_Pkg is
-- First, define the basic units of the problem:
  type Fuel_Units is ( Pounds_Fuel,
                       Nautical_Miles,
                       Hours );
-- Then instantiate a copy of Units_Integrity for the fuel problem:
  package TDA_Units_Integrity is new Units_Integrity ( Numeric_Value => Float,
                                                         Units_Type=>Fuel_Units );
  use TDA_Units_Integrity; -- to get this example on a single page.
-- The variables involving units are:
  Fuel_On_Board,
  Fuel_Rate,
  Interceptor_Speed,
  Mission_Duration,
  Mission_Distance : Physical_Quantity;
-- ...
begin -- TDA_Overview_Pkg
-- Lock in the exponents of the units of the variables:
  Fuel_On_Board      := Init ( Numerical_Value => Initial_Fuel,
                               Units           => ( Pounds_Fuel    => 1,
                                                       Nautical_Miles => 0,
                                                       Hours           => 0 ));

  Fuel_Rate          := Init ( Numerical_Value => Known_Fuel_Rate,
                               Units           => ( Pounds_Fuel    => 1,
                                                       Nautical_Miles => 0,
                                                       Hours           => -1 ));

  Interceptor_Speed  := Init ( Numerical_Value => Known_Interceptor_Speed,
                               Units           => ( Pounds_Fuel    => 0,
                                                       Nautical_Miles => 1,
                                                       Hours           => -1 ));

  Mission_Distance   := Init ( Numerical_Value => 0.0,
                               Units           => ( Pounds_Fuel    => 0,
                                                       Nautical_Miles => 1,
                                                       Hours           => 0 ));

  Mission_Duration    := Init ( Numerical_Value => 0.0,
                               Units           => ( Pounds_Fuel    => 0,
                                                       Nautical_Miles => 0,
                                                       Hours           => 1 ));

-- ...
-- After the Mission_Distance has been computed, find the Mission_Duration:
  Assign ( Source => Mission_Distance / Interceptor_Speed,
           Target => Mission_Duration );

-- Then adjust the Fuel_On_Board to predict Fuel_On_Board when the mission is
-- completed:
  Assign ( Source => Fuel_On_Board - Fuel_Rate * Mission_Duration,
           Target => Fuel_On_Board );
-- ...
end TDA_Overview_Pkg;

```

ADA SOFTWARE REUSE IN SUPPORT OF OPERATION DESERT STORM

Russell J. Brown and Judy L. Morgan

Merit Technology Incorporated
5068 West Plano Parkway
Plano, Texas 75093

Summary

During Operation Desert Storm, Merit Technology participated in an effort led by Lawrence Livermore National Laboratory to construct a ballistic missile early warning/analysis system. The effort was driven by a need to graphically display and reconstruct ballistic missile attacks carried out by Iraq to determine doctrine and assess effectiveness of the Iraqi ballistic missile program. For the system to be constructed in the necessary time frame, existing software had to be quickly combined and modified to create the application. Merit had recently finished an air situation display for the government which combined a graphical user interface with two-dimensional mapping capability on an advanced graphics workstation. All of the software had been written in Ada. A small development team combined COTS software with software from the existing program to create a system to display ballistic missile trajectory data in 2D and 3D in a matter of weeks. Approximately 90% of the estimated 30,000 lines of Ada code were reused. All who were involved with the effort, from industry and government, proclaimed the system overwhelmingly successful, and a testimony to the "ilities" of the Ada language.

Problem

When Operation Desert Shield erupted into Operation Desert Storm in January of 1991, Saddam Hussein of Iraq began a campaign of terrorist warfare against the countries of Israel and Saudi Arabia. The principal terrorist weapon used was the Soviet-made SCUD short-range ballistic missile. Although many of the permanent missile silos were destroyed early in

the campaign, the elusive mobile launchers continued to evade Coalition forces and carry out missions against friendly nations. The need to analyze the Iraqi tactics and strategy of the SCUD campaign as it was executed became apparent. Studying the missile attacks and results as they occurred would help Coalition forces anticipate impending launches and retaliate in a timely manner. The Strategic Defense Initiative Office (SDIO) tasked the Lawrence Livermore National Laboratory (LLNL) at the University of California to quickly create a prototype of a distributed, multi-user system to provide theater-level SCUD missile early warning and analysis functions. Construction of such a system would entail combining available data resources, both space-based and terrestrial with the latest 2D and 3D graphics display technology. The prototype system would be installed at Air Force Space Command at Peterson Air Force Base in Colorado Springs, Colorado.

Available Resources

From their extensive involvement with SDIO programs, LLNL had personnel with expertise in handling the data from the sources necessary for the prototype system. They began seeking existing technology in the areas of advanced 2D and 3D graphics, along with Man-Machine Interface expertise. Through reference material provided by Silicon Graphics, Inc., the manufacturer of the workstations to be utilized, they discovered Merit Technology, Inc. In December of 1989, Merit had successfully completed a project for Air Force Aeronautical Systems Division (ASD) at Wright-Patterson Air Force Base in Dayton, Ohio. The project combined the display of air situation data on a variety of map backgrounds with advanced MMI

technology, all implemented in Ada. The task at hand was to quickly combine the available resources to produce a functional prototype in time to contribute to the war effort.

Strategy

In order to meet the imposed time constraints, commercially available hardware and commercial-off-the-shelf (COTS) software to the maximum extent possible. The LLNL/Merit team immediately met for an intensive requirements analysis/design session. The Ada software from the recently completed Air Force ASD project was utilized to complete the Air Force Space Command effort. The baseline software had to be modified in some areas and extended in others to accommodate the identified requirements for the prototype system.

Implementation

The baseline software consisted of approximately 30,000 lines of Ada software, developed over a period of two years for the ASD project. Object-Oriented requirements analysis and design/development techniques had been implemented during the life of the project. The resulting Ada software was maintained in a Revision Control System software repository.

The highest level object view of the baseline software is diagrammed in Figure 1. The COMMUNICATION object provided the vehicle to transmit and receive data across a local area network. The DATA object provided the formats of the data messages to be received and transmitted by the system. The SITE, TRACK, and LINK objects were self-contained packages which were driven by the incoming data. The TEXT object was responsible for the manipulation and display of text embedded within the incoming messages. The OVERLAY object provided an interactive graphical tool for the user to create, save, and modify "grease pencil" annotations on the display. The 2D MAP object consisted of a proprietary COTS software package written in C and the required Ada bindings to these routines, providing 2D display windows composed of Defense Mapping Agency (DMA) mapping data. Digital Terrain Elevation Data (DTED) and Arc-second Raster Chart Digitized Raster Graphic (ARCDRG) digitized

paper map data (including Landsat and SPOT imagery) provide the map backgrounds, while data such as the CIA World Data Bank II (borders and coastlines) and Digital Feature Analysis Data (DFAD) served as overlay data on the map. The Man-Machine Interface (MMI) object provided the system interface to the user. The interface was completely defined in ASCII data files allowing "look and feel" changes without code modifications.

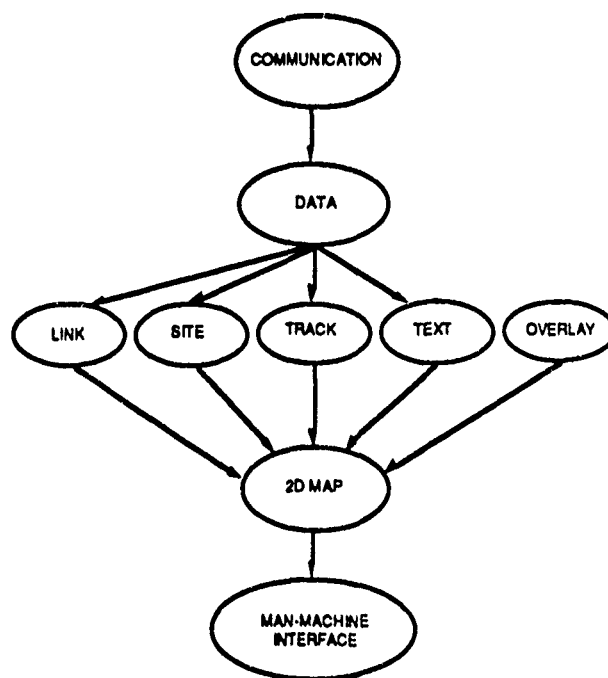


FIGURE 1. EXISTING SOFTWARE ARCHITECTURE

Due to the utilization of the concepts of OORA/OOD combined with the Ada programming language, the existing software was easily modifiable to meet the task at hand. The COMMUNICATION object was modified to accommodate the distributed network architecture. The DATA object was changed to handle the new missile trajectory messages passed to the system from external data sources. The SITE and LINK objects were simply deleted, since they had no bearing on the new type of data driving the system. The TRACK package had to be changed from dealing with aircraft data to dealing with SCUD ballistic missile data. The OVERLAY function was reused without modification. Since the MMI object was ASCII file-driven, the software was used without modification. The only change

necessary was in the text files which defined the functions available to the user. The existing system did not deal with 3-Dimensional data displays. Integrating this capability consisted of plugging in a COTS software package and writing 3D rendering functions for the new TRACK data. The resulting software architecture is illustrated in Figure 2.

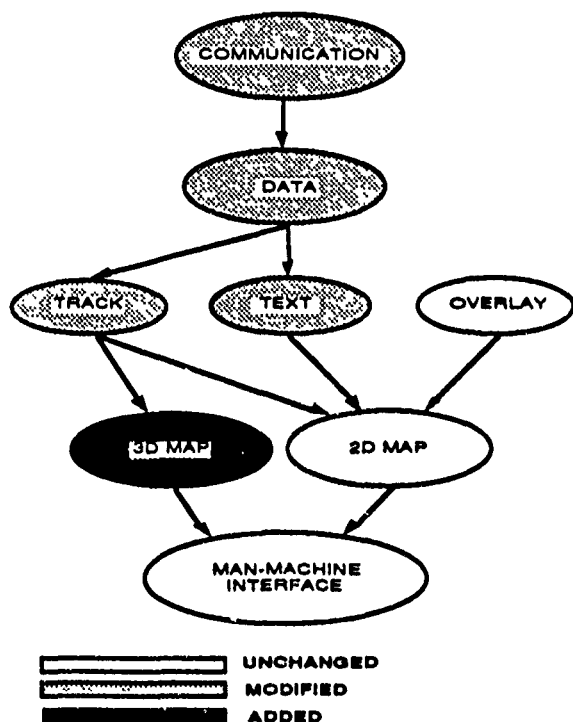


FIGURE 2. REVISED SOFTWARE ARCHITECTURE

Additional effort involved processing every map that the Defense Mapping Agency had for the theater of interest. Nine different scales of maps were processed by either manual scanning/warping or by processing of CD ROM's provided by DMA.

Results

Combining the efforts of less than a dozen engineers with commercially available hardware, COTS software, and existing software from an Ada software repository, the LLNL/Merit team produced a working prototype of the required system in a matter of six weeks. The system includes six commercially available graphics workstations linked on a distributed ethernet

network. Incoming data is broadcast to each display in the system. The workstations on the network have the capability to send and receive messages among themselves, to avoid contention over incoming data. This also allows monitoring of each station's activity from a "master" station. To facilitate operator learning, the system was built to handle training and live mission modes. Automatic switch to live mode occurs if actual data is received at any time.

By displaying current and historical missile data along with user-created overlays, a comprehensive picture of the missile campaign can now be drawn. Tactics can be analyzed in new and different ways, with the 3D terrain analysis providing valuable launch and escape route information. The capability also exists to compare various trajectory prediction algorithms and the reactions of the Allied forces against known data and results.

According to DMA, another result of the system is the largest digitized map and multi-spectral imagery database in existence. Forty gigabytes of disk space was utilized to accommodate the processed map data.

Lessons Learned

The results achieved in this effort would not have been possible just a few short years ago. The development of hardware and software has been revolutionized. The concepts of "open systems" and "reusable software" were unheard of until relatively recently. The growing use of the Ada programming language is a major factor in this revolution. If the software written for the ASD project had not been designed and implemented using modern tools such as Ada, achieving the success of this effort would have been extremely difficult. However, design and implementation are only part of the battle. The best software in the world cannot be reused if it is not documented and archived in known repositories. The more that government and commercial bodies contribute to the area of software reuse, the less software development will cost in the future.

Russell Brown received a Bachelor of Science degree in Computer Science from North Texas State University in December of 1984. From 1985 to 1987 he served as a Software Engineer at E-Systems Greenville Division designing, developing, testing and integrating real-time software systems for airborne and ground-based intelligence collection/processing systems. In 1987, he joined Electrospace Systems, Inc. as a programmer analyst, designing and developing Ada software for the Strategic Air Command's National Emergency Airborne Command Post. From 1989 until the present, he has been working in the Intelligence and Space Systems Division of Merit Technology, Inc., serving as a Member of Technical Staff. Duties have included preparation and presentation of Ada/OOD in-house training seminars, Ada software system design and implementation, and more recently, program management and marketing responsibilities.

Judy Morgan graduated from the University of Sciences and Arts of Oklahoma in April of 1985 with a Bachelors of science degree in Computer Science and Mathematics. From 1985 to 1987 she worked as a Software Engineer at Rockwell International on an embedded real-time radio receiver in written Ada. From 1987 until the present she has been a Member of the Technical staff in the Space Systems Division of Merit Technology, Inc. Her responsibilities include software system design, implementation and integration.

Development of Cost Estimation Prototypes

A J C Cowderoy and J O
Jenkins
School of Informatics, City
University
London, EC1V OHB
United Kingdom

Abstract -- Many organisations fail to realise the budgets and deadlines they have set for software development projects. Research has shown that the accuracy of the predictions made using the current generation of cost estimation methods is low.

The ESPRIT project, MERMAID, addresses these issues. Its principal research objective is to improve understanding of the relationships between the measureable attributes of software development projects. In addition it plans to deliver a series of prototype cost estimation toolkits. To date two such prototypes have been delivered.

The MERMAID project began in late 1988 and has developed a new cost estimation methodology. This is based on the use of

locally defined metrics. Both Statistical and Analogical Methods are used to generate forecasts (estimates). A fundamental tenet of the MERMAID approach is to avoid the use of estimates as the independent variables in statistical forecasting.

The first prototypes provided facilities for product sizing as well as effort forecasting. The next major release due late in 1992 will, in addition, offer risk assessment and resource modelling facilities. The latter is of special note in that the project is investigating a number of novel approaches including Systems Dynamic Modelling.

The paper will provide an architectural overview of the planned toolset as well as summarising the results of empirical research.

Index Terms -- Cost Model, Software Metrics, Risk Assessment.

1. INTRODUCTION

The tendency for software development project to be completed over schedule and over budget has been

documented extensively 1, 2. Additionally many projects are completed within budgetary and schedule target only as a result of the customer agreeing to accept reduced functionality.

A particular area of research of relevance to this phenomenon is software cost modelling. Many researchers have attempted to model the interrelationships of a project cost for instance Putnam³. These parameters are the Total Project Effort (in person time units), Elapsed time or schedule, T and the average staffing level M throughout the project. It might be assumed that there was a simple relationship between the three viz:

$$E = T.M \quad [1]$$

In his classic book, The Mythical Man Month, Fred Brooks⁴ exposes the fallacy that effort and schedule are freely interchangeable. All current cost models are produced on the assumption that there is very limited scope for schedule compression unless there is a corresponding reduction in delivered functionality.

2. MERMAID

The Metrication and Resource Modelling Aid (MERMAID) project, partially financed by the Commission of the European Communities (CEC) as Project 2086 began in October 1988 and its goals are as follows:

- Improvement of understanding of the relationships between software development productivity and product and process metrics
- To facilitate the widespread technology transfer from the Consortium to the European Software Industry
- To facilitate the widespread uptake of cost estimation techniques by the provision of prototype cost estimation tools.

The applicability of the tools developed by the Mermaid consortium is considered to encompass both embedded systems and Management Information Systems.

Mermaid has developed a family of methods for cost estimation, many of which have had tools implemented in the first two prototypes. These prototypes are best

considered as toolkits or workbenches. Figure 1 gives an architectural overview of these prototypes.

The first prototype was demonstrated in November 1990. It was developed on a SUN 3/60 C Workstation using an objective oriented extension of the C language, Objective C. Two versions exist, one running under the Portable Common Tool Environment (PCTE) and the other is a UNIX implementation. The second prototype was demonstrated in November 1991, versions of which were developed on an IBM PS/2 running either WINDOWS 3 or OS/2 and Presentation Manager.

3. MERMAID ESTIMATION METHODS

Before describing the functionality implemented in the prototypes, an explanation of the MERMAID methodology is required. At the time of the start of the MERMAID project, October 1988, the commonly used approaches to cost estimation were as follows:

- Expert Judgement, ie informal guestimate of the resources required
- Analogy, similar to the above but influenced by the identification of a similar

completed project, similar to the one being planned.

- Parametric Model, use of a cost estimation tool based on a number of existing models of the relationships between project cost parameters and cost drivers. Models on which tools were based included SLIM³, COCOMO⁵, AND COPMO⁶.

Tools based on these models can be calibrated for a particular environment. However, research has shown that despite calibration, the accuracy of estimates produced by cost estimation tools is poor⁷. There are several contributing factors to this inaccuracy. These include the difficulty, if not impossibility, of estimating the size of the product to be developed as early in the lifecycle as the Requirement Analysis Phase. Additionally calibration depends on the existence of moderate quantities of past project data collected in a consistent manner.

The MERMAID approach is based on the use of locally-based and user-defined attributes and metrics. Furthermore wherever possible actual measures as

against estimates are used as input to the estimating facilities. Today's tools based on parametric models normally require the project manager or estimator to input an estimate of the size of the software product to be developed. This is either expressed in Lines of Code (LOC) or in the form of a function-based metrics, Function Point Count. This latter metric was developed within IBM by Alan Albrecht⁸ and purports to measure the size of an application in terms of the functionality to be delivered to the user. It has the advantage over LOC in that it can be measured as soon as an outline logical design of the application is available. The development lifecycle model assumed by MERMAID, is that a project is regarded as a series of milestones separated by phases. This view enables the estimator to model any organisations lifecycle. Care must be taken not to confuse this use of the word phase with its use in the Waterfall lifecycle model.

4.DEVELOPMENT PRODUCTIVITY ANALYSIS

Bailey and Basili⁹ suggested that it should be possible to develop a satisfactory

estimation model for given environment using a small number of independent product and process attributes as the dependent variables in a multivariable linear regression model. Such an approach differs from that found in the COCOMO⁵ model where the nominal estimate derived from the standard estimation model is adjusted by a multiplicative adjustment factor. This measures the combined influence of a number of cost or productivity drivers. Recent research¹⁰ has suggested that many these cost drivers are not independent.

An analysis by MERMAID¹¹ of a dataset from various commercial MIS environments using principal component analysis of 21 supposed productivity factors indicated that 7 principal components accounted for over 75% of the variability of the data and that no other component accounted for more than 5% of the variability. The main source of the belief that staff and environment characteristics are significant factors in determining development

productivity is the importance placed on them in most of the published cost models. A move of caution against reading too much into these analyses must be sounded. Any retrospective validation using data from completed projects is fraught with methodological difficulties, not the least being the uncertainty over the accuracy of the measurements. Classical experimental design involving the monitoring of on-going projects is rarely possible in this domain.

5. MERMAID AND ESPRIT

The Commission of the European Communities (CEC) launched ESPRIT in 1983 largely as a consequence of the realisation in Europe that its Information Technology industry was becoming increasingly uncomparative particularly viz-a-viz Japan. ESPRIT projects cover both hardware and software technologies and address both factory and office automation. Both software engineering and artificial intelligence are primary focuses. MERMAID is one of a number of projects, some current and others which

have been completed, which address aspects of software development, management and metrics. Two immediate predecessors of MERMAID in which some of ideas were developed were the ESPRIT projects IMPW¹² and REQUEST¹³. The former developed an integrated project management toolkit which included support for cost estimation and latter was concerned with modelling aspects of software quality and reliability.

ESPRIT has finally begun its third phase and today's projects have a clearer end user requirement orientation than those in the first phase which were largely technology push projects. The change in emphasis followed early evaluations of the programme which showed a disappointing take-up of the research output by industry even in cases where industry itself was the prime mover behind the project. Nevertheless, the CEC's Court of Auditors have said "The industrial results of the ESPRIT programmes have already achieved a scale that is well beyond that of the other (CEC) research programmes" To date over 800 organisations

have participated in 480 projects which had produced over 300 significant results by the start of this decade. Over \$5 billion has been allocated for ESPRIT funding between now and the end of the decade. This will be matched by an equal amount of industrial funding.

6. THE IMMEDIATE FUTURE

The MERMAID consortium is putting the finishing touches to the specification of the tool functionality to be included in the Mark 2 prototype due for release late in 1992. Considerable attention is being paid to developing a risk assessment capability which conforms to the MERMAID philosophy. Such a risk assessment capability will require access to a knowledge base of previous projects.

and measures of risk to project budget and schedule will be estimated using similar statistical and analogical techniques to those used for effort estimation. In this context the risk driver is viewed similarly to a cost driver. In addition the inclusion of a facility to examine effort and schedule trade-off is planned. This will be based

on the Kunomaa Resource (KURE) Model which takes a thermodynamic view of the software development process. An alternative to this involves the use of System Dynamics Models¹⁴. Superior sensitivity analysis capability can be provided in this way provided an adequate understanding of the relationships between schedule effort and manpower level is established for the development environment. No decision as the final makeup of the functionality of the Mark 2 prototype has been made at the time of writing of this paper.

LIST OF REFERENCES

1. de Marco T (1982) Controlling Software Projects, Yourdon Press, New York.
2. Congress (1989) Bugs in the Program: Problems in Federal Government Computer Software Development and Regulation. Staff Study for the House of Representatives Committee on Science, Space and Technology.
3. Putnam L H (1987). A general empirical solution to the macro software sizing and estimation problem IEEE Trans.

Software Engineering SE-4
(4)

4. Brooks F (1975) The
Mythical Man Month,
Addison-Wesley, London

5. Boehm, B W (1981)
Software Engineering
Economics, Prentice Hall,
Englefield Cliffs, N J

6. Conte S, Dunsmore H and
Shen V Y, (1986), Software
Engineering Metrics and
Models, Benjamin-
Cummings, Menlo Park CA.

7. Kemerer C F (1987), An
empirical Validation of
Software Cost Estimation
Models, Comm. ACM Vol
30(5).

8. Albrecht A J, Measuring
Application Development
Productivity Proceedings
Joint SHARE/GUIDE
Symposium, October
1979, pp83-92.

9 Bailey J W and Basili V
(1981) A Meta model for
software development
resource

expenditure. Proceedings of
the 5th International
Conference on Software
Engineering, pp107-116.

10. Subramian G H and
Breslawski S (1989) A Case
for dimensionality reduction
in software development of
effort estimates. TR 89-02
Computer and Information
Science Department, Temple
University, Philadelphia PA

11. Kitchenham B and

Kirakowski J (1991). 2nd
Analysis of Mermaid Data.
Mermaid Project Deliverable
D 3.3.B.

12. Bosco M, Jenkins J and
Verbruggen R
Integrated Management
Process Workbench (IMPW).
Advance Working Papers, 1st
International Workshop on
CASE, Cambridge, Mass.

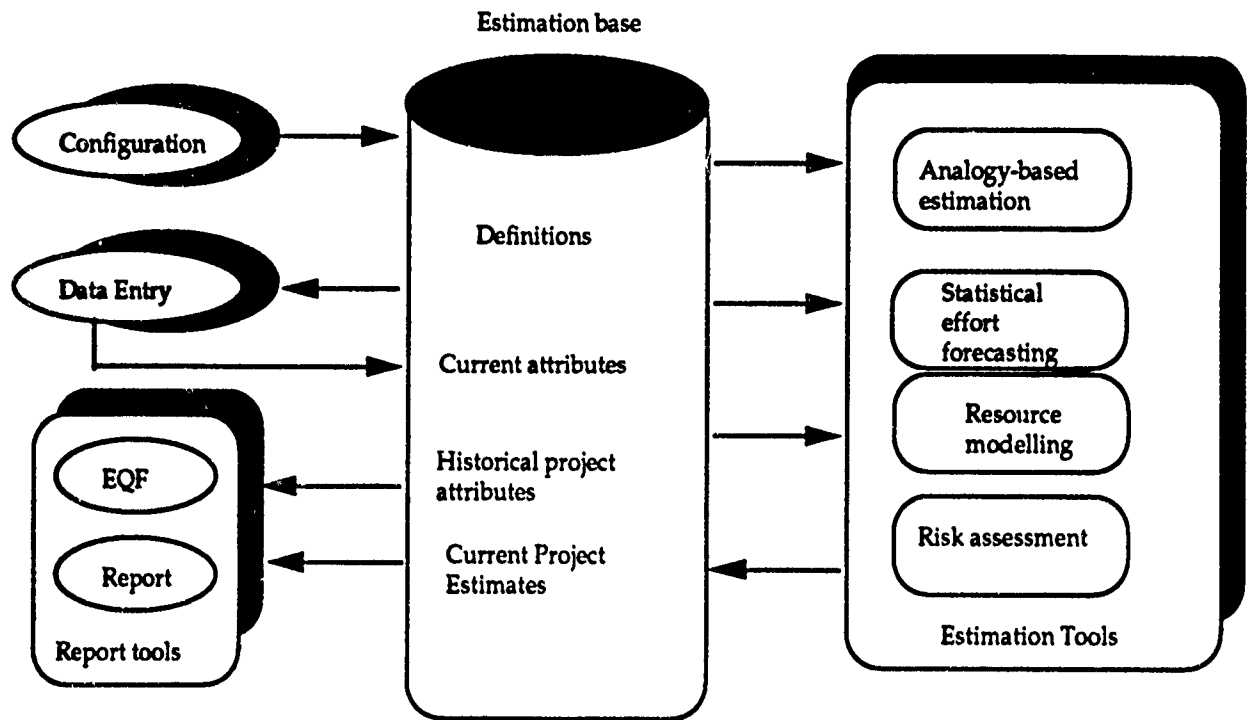
13. Linkman S J (1990)
Quantative monitoring of
software development by
time-based and
intercheckpoint
monitoring. Software
Engineering Journal Vol5(1)

14 Harry, C M and Jenkins J
(1991) Mermaid Working
Paper TCU-5-0-D-500/1

15. Bell G and Jenkins J
(1991),
Proceedings of the 7th
International COCOMO Users
Group Meeting, SEI, Pittsburgh.

FIGURE 1

Mermaid 1 architecture



ARTIFICIAL INTELLIGENCE PANEL

Chairperson: Dr. Richard Kuntz, Monmouth College

THE DEVELOPMENT AND APPLICATION OF AN ADA EXPERT SYSTEM SHELL

Dr. Verlynda S. Dobbs

C. Alan Burnham

Telos Corporation
55 N. Gilbert Street
Shrewsbury, New Jersey 07702

ABSTRACT

This paper describes the development and verification of an experimental Ada EXpert System Shell (AXS) and its use to demonstrate the possibility of using an Ada-based expert system for processing tactical message data. Keywords: artificial intelligence in Ada, expert systems, and metric evaluation.

progress and many successes have been reported. [12,9,14,1,8,13]

This paper describes the development and verification of an experimental Ada EXpert System Shell (AXS) and its use to demonstrate the possibility of using an Ada-based expert system for processing tactical message data.

INTRODUCTION

While software engineering with Ada has been evolving to aid software developers and supporters, artificial intelligence (AI) technology has become increasingly prevalent. AI offers tremendous potential in DOD systems, especially in dealing with the information overload facing DOD systems operators. Expert system technology offers the most advanced and most widely used AI technique to aid in sorting and interpreting the data presented to the operator. Expert systems are developed to represent and apply factual knowledge in a specific domain (i.e. DOD systems). Unfortunately, the use of expert systems by DOD has been limited. This limitation has been partially due to the problems of reliability and maintainability surrounding systems implemented in traditional AI languages.

Expert systems are commonly developed with the aid of expert system shells - software tools used to aid the knowledge engineer in coding and executing knowledge. Currently, most shells are developed in either Lisp or C. Only a small amount of commercial effort has been expended to create shells in Ada. Although Ada may not be the ideal language for creating expert systems, Ada does offer extensive benefits, especially when the software is to be integrated and maintained with embedded, real-time DOD applications. At annual conferences on Artificial Intelligence and Ada, much

AXS DEVELOPMENT

In the sections below are discussions of AXS design goals, AXS components, AXS development environment, and AXS design goals verification.

DESIGN GOALS

The use of Ada for system development encourages good software engineering practices. With this in mind, design goals for AXS were chosen as: portability, modularity, maintainability, and flexibility. Success in meeting these goals was verified through metric analysis of the source code and demonstration.

AXS COMPONENTS

The basic components of AXS, described below, are the knowledge base structuring mechanism, the inference engine, the development interface, and the explanation facility. The AXS components are shown in Figure 1.

The control of the system is incorporated into the knowledge base structure and the inference engine. The knowledge base contains the information needed by the inference engine to create the expert system. The knowledge base is divided into frames that represent objects and rules that indicate actions. The frames contain slots that store information about the object.

These slots have values and sets of rules (called demons) attached to them. When the type of access to a slot (value added, value needed, etc.) matches the type of demon (if-added, if-needed, etc.) the rules associated with that demon are fired.

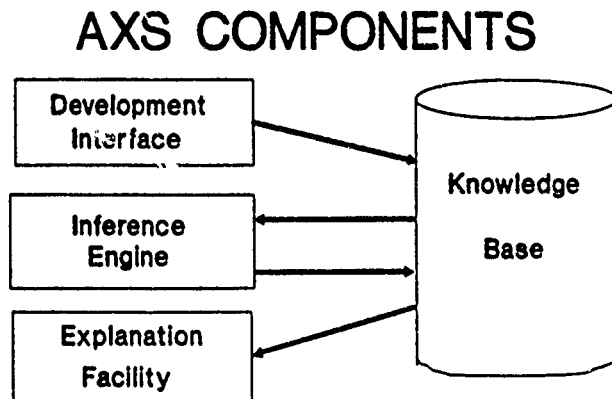


Figure 1

The inference engine provides both forward and backward inferencing methods. To implement these methods, each frame is classified as a start frame, an intermediate frame, or a goal frame. Pure forward inferencing begins processing by trying to fill in all values in the start frames and continuing the inferencing until no more inferences can be made. Pure backward inferencing begins by attempting to fill all the slots in any goal frame by backward chaining of rules.

The development interface provides the means for an expert system designer to construct a knowledge base to be run by the inference engine.[4] First priority for the AXS development interface has been given to implementing as generic an interface as possible, one that can be supported by any Ada compiler. A menu system is used for inputting data from the keyboard. From the menu, the developer can choose to create frames, rules, or the end-user interface. The development interface relieves the developer of the details of the syntax of the knowledge base file.

The explanation facility (EF) serves as an explanation of the expert system for users, a debugging tool for developers, and as a teaching/tutorial tool for new users.[11] The AXS EF provides both runtime explanations and end-of-processing explanations. The runtime EF includes explaining:

- the contents of a rule
- why information is being requested
- how the system arrived at a certain point

- the current value of a slot, and
- relationships of objects and attributes.

The end-of-processing EF provides functions 1, 4 and 5 above in addition to displaying both the critical path and the execution trace. For use in embedded systems, the entire EF can easily be unplugged from the expert system by simply commenting out 15 lines of code and removing the EF package from the compilation order.

DEVELOPMENT ENVIRONMENT

The object-oriented design for AXS[3] was implemented using Verdex Ada in a Unix environment on a Sun workstation. In previous work, AXS was used to successfully implement a classification system that identified aircraft.[2] AXS is currently being enhanced for use at Wright Patterson Air Force Base (WPAFB), Dayton, Ohio.

VERIFICATION OF DESIGN GOALS

Verification of the degree to which AXS met its design goals of portability, modularity, maintainability, and flexibility, was accomplished by using a mix of metric analysis and demonstration.

Each design goal, along with the method/methods used to evaluate it, is shown in Table 1.

Table 1
AXS DESIGN GOALS

Portability	Metric Analysis Build Expert Systems In Two Environments
Modularity	Metric Analysis
Maintainability	Metric Analysis Effort Required to Maintain/Upgrade
Flexibility	Develop Different Types of Expert Systems

The metric analysis was accomplished using AdaMAT [6,7,10] to evaluate the AXS source code. AdaMAT is a commercially available metric tool developed by Dynamics Research Corporation, that uses over 400 metrical elements. These elements can be combined to form overall aggregate results and calculate metric values for software characteristics such as: independence,

modularity, simplicity, system clarity, and maintainability.

Definitions of these characteristics are shown below along with typical ranges of scores from other projects:

Independence (or Portability) - Those attributes of the software that determine its non-dependency on the software environment (computing system, operating system, utilities, input/output routines, libraries). Typical scores are .85 to .99.

Modularity - Those attributes of the software providing a structure of highly cohesive modules with optimum coupling. Typical scores are .20 to .60.

Simplicity - Those attributes of the software providing for the definition and implementation of the functions of a module in the most non-complex and understandable manner. Coding simplicity, design simplicity, and flow simplicity are considered. Typical scores are .30 to .40.

System Clarity - Those attributes of programming style providing for a clear and understandable description of the program structure. Typical scores are .30 to .80.

Maintainability - A roll-up of the above four measures to provide an overall assessment of the ease with which the code can be subdivided, understood, enhanced, and modified. Typical scores are .40 to .70.

The results of using AdaMAT to evaluate the AXS source code are: independence - .96, modularity - .65, simplicity - .55, system clarity - .77, and maintainability - .75. These scores are shown in comparison with the typical low and high scores in Figure 2.

The AdaMAT scores for AXS all verified that AXS was successful in meeting its design goals. The portability of AXS was verified by the high value of the independence metric calculated by AdaMAT. Portability was further demonstrated by using AXS to develop the application described below. The AdaMAT modularity and maintainability scores for AXS exceeded the typical high scores, supporting the success of AXS in meeting the goals of developing a highly modular, maintainable system. The maintainability goal was additionally demonstrated by the ease with which various developers, over a three-year period, could use, modify, and upgrade AXS. The flexibility goal

AdaMAT Results

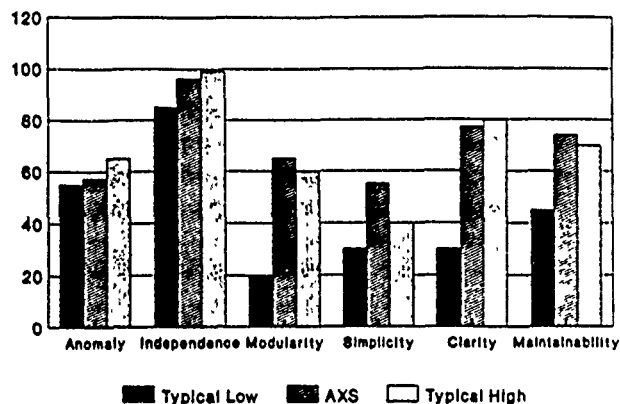


Figure 2

was demonstrated by the use of AXS to develop a classification expert system to identify aircraft and to develop the expert system described in the following sections.

AXS APPLICATION - COMDES

The Correlation of Message Data Expert System (COMDES) COMDES is an experimental expert system that was created to explore the possibility of using an Ada expert system shell to correlate tactical message data. Developing COMDES also demonstrated the portability and flexibility of AXS. The sections below describe the COMDES problem domain, the knowledge acquisition process, the knowledge base development, and the results of executing the expert system.

DOMAIN DESCRIPTION

Today's semiautomated battlefield information systems are essentially message exchange systems. They transfer data via structured message formats having both fixed coded message fields and free format ASCII text fields. During an engagement, battlefield Tactical Operation Centers (TOCs) are essentially in information overload. Expert system technology is currently being investigated to aid operators in more efficiently processing these messages.

One area of heavy message traffic is that of fire support. Because of the availability of information and interest in this area, a set of realistic tactical artillery messages was selected as the basis for the domain for the creation of COMDES.

KNOWLEDGE ACQUISITION

The process of knowledge acquisition for this project began with on-site discussions with the user at Ft. Sill, Oklahoma. In support of development work for the Advanced Field Artillery Tactical Data System (AFATDS), the Field Artillery Board at Ft. Sill prepared an extensive set of realistic tactical artillery messages in TACFIRE format, based on specific Operation Orders for a given Force Structure. The set of messages is referred to as a Time-Ordered Event List (TOEL) and consists of well over 2000 tactical messages. For this study a subset of messages between selected units for a 100 minute interval was extracted from the TOEL. To better focus this study only the Artillery Target Intelligence (ATI) messages were used. The ATI messages were chosen because methods for correlating the data contained in these messages are fairly well understood and the message content is similar to message data that may be exchanged between other users.

ATI messages are messages that provide information (type, location, strength, time, etc.) about enemy units. ATI messages are analyzed, then, based on certain criteria, combined for purposes of fire support engagement. Whether or not targets are combined is a function of the type, location, and time of observation.

The general rules for the correlation and fusion of the target data were obtained from various Field Artillery School manuals and discussions with users. The subset of messages was manually decomposed and reformatted for ease of use and understanding.

As a result of the knowledge acquisition, three factors for combining/correlating potential targets were identified: type, distance and time. The type correlation is the probability that two targets of different types would be combined if distance and time were not a factor. The distance correlation is the maximum distance by which the targets can be separated and still be considered for combining. The time correlation is the maximum time that can elapse between target sightings and still be considered for combining.

KNOWLEDGE BASE DEVELOPMENT

After the knowledge acquisition phase was completed, the knowledge base had to be built to represent the knowledge that was acquired from the user. The first action was to identify the types of frames that would

be appropriate to interpret and correlate the ATI message data. The two basic types of frames both represented target data. One type dealt with generic target correlation data, while the other type represented specific targets extracted from the ATI messages. Next the specific frames and their associated slots were identified and implemented. Then the appropriate demons and the rules were associated with each of the slots.

COMDES EXECUTION RESULTS

COMDES was executed using the information from the messages extracted from the TOEL. The ATI message processing expert system uses its frames and rule base to compare target information from ATI messages to determine if potential targets can be combined, thereby simplifying target correlation for the user. The output, which is currently presented to the user in textual format, was successfully verified by comparison with anticipated results.

COMDES demonstrated that the AXS was, in fact, portable. COMDES was created on a HP 9000 system, and the only effort involved in porting from the Sun workstation to the HP system was to rename the Ada files to be compatible with the CHS and recompile the source code. By using AXS to build COMDES, we were also able to demonstrate that AXS was sufficiently flexible to allow the creation of other than classification expert systems.

FUTURE EFFORTS

There are several areas in which additional work could be performed to complement the work accomplished so far. One area is that of providing enhancements to the AXS user interface through the use of graphical capabilities, possibly by the use of X Windows. Another area would be the enhancement of the functional capability of the AXS library, providing AXS with a more robust shell that could be more readily adapted to other applications. A third area for improvement would be through optimization of the expert system through reuse of rules and use of Ada compiler options to decrease memory requirements and execution time.

In addition to the above enhancements, a comparison of AXS with other existing Ada expert system shells, in terms of ease of use and functional capability, would provide increased insight into the current potential for using Ada in developing expert systems for DOD.

ACKNOWLEDGEMENTS

Portions of this effort were performed by Telos Corporation under Contract Number DAAB07-89-D-A050, for the U.S. Army Communications Electronics Command Center for C3 Systems; and by Wright State University as a subcontractor to Telos Corporation. Portions of AXS were developed by Wright State University as a subcontractor to SAIC for WRDC/AAWA-1 of the U.S. Air Force, WPAFB, Ohio.

AdaMAT is a registered trademark of Dynamics Research Corporation.

REFERENCES

- [1] M. Adkins. Flexible data and control structures in ada. In *2nd Annual Conference on Artificial Intelligence and Ada*, pages 9.1-9.17, 1986
- [2] J. Cardow. Toward an expert system shell for a common ada programming support environment. Master's thesis, Wright State University, 1989.
- [3] J. Cardow and V. Dobbs. Toward an expert system shell for a common ada programming support environment. In *NAECON-89*, pages 1042-1047, 1989.
- [4] J. Courte and V. Dobbs. A development interface for an expert system shell. In *8th Annual National Conference on Ada Technology*, pages 623-632, 1990.
- [5] V. Dobbs and C. A. Burnham. Correlation of target message data using an ada expert system shell. Technical Report U.S. Army Contract Number DAAB07-89-D-A050, Telos Corporation, 1990.
- [6] Dynamics Research Corporation, Andover, MA. *AdaMAT Reference Manual*, 1988.
- [7] S. Keller and J. Perkins. Ada measurement based on software quality principles. In *Washington Ada Symposium*, pages 195-203, 1985.
- [8] D. LaVallee. An Ada inference engine for expert systems. In *First International Conference on Ada Programming Language Applications for the NASA Space Station*, pages E.4.3.1-E.4.3.12, 1986.
- [9] S. D. Lee. A distributed architecture for real-time expert systems. In *6th Annual Conference on Artificial Intelligence and Ada*, pages 33-50, 1990.
- [10] S. Levine, J. Anderson, and J. Perkins. Experience using automated metric frameworks in the review of ada source for afatds. In *Proceedings of the 8th Annual National Conference on Ada Technology*, pages 597-612, 1990.
- [11] V. Saunders and V. Dobbs. Explanation generation in expert systems. In *NAECON-90*, 1990.
- [12] A. Wallen and S. Lubash. A high performance ada-based real-time embedded expert system shell. In *6th Annual Conference on Artificial Intelligence and Ada*, pages 21-32, 1990.
- [13] K. et. al. Wallnau. Construction of knowledge-based components and applications in ada. In *4th Annual Conference on Artificial Intelligence and Ada*, pages 3.1-3.21, 1988.
- [14] P. Wright. Ada real-time inference engine. In *5th Annual Conference on Artificial Intelligence and Ada*, pages 83-93, 1989.

AUTHORS

Verlynda S. Dobbs is a senior computer scientist with Telos at Ft. Monmouth, NJ. She is currently developing a Network Planning Tool for military communication systems. Dr. Dobbs was formerly on the faculty of the Department of Computer Science and Engineering at Wright State University. Her research interests are in the areas of software engineering, artificial intelligence, and Ada for artificial intelligence. Dr. Dobbs received her Ph.D in computer science from the Ohio State University.

C. Alan Burnham is a senior systems engineer with Telos at Ft. Monmouth, NJ. He has worked for over 20 years in the development and support of software-intensive military systems. Mr. Burnham's recent experience has been primarily in developing specifications, defining requirements, evaluating design, and testing software for mission-critical defense systems. His earlier experience was in the development, validation, and application of large-scale, high-resolution combat simulations. Mr. Burnham received his BA in Mathematics from Augustana College.

BOILERMODEL: A QUALITATIVE MODEL-BASED REASONING SYSTEM IMPLEMENTED IN ADA

James F. Stascavage and Yuh-jeng Lee

Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

Effective, inexpensive, and realistic on-going training is required to keep all Naval personnel proficient in their fields. Nowhere is this more true than in steam propulsion engineering plants. The complex systems of valves, piping, and components require continual refresher for watchstanders to perform their jobs safely. BoilerModel is a qualitative expert system designed using model-based reasoning principles and implemented in Ada. It accurately models a 1200 psi D-type boiler and its associated peripherals. The use of fundamental intra-component relationships ("first principles") and constraint propagation result in compact code because there is no need for the extensive rule base found in conventional expert systems. Implementation in Ada permits the use of concurrent tasking to simulate simultaneous valve propagation found in real-world boiler systems. Additionally, Ada's portability allows BoilerModel to be compiled and run on virtually any machine, thereby making it an affordable and attractive complement to shipboard engineering training.

INTRODUCTION

The technical nature of most U.S. Navy jobs requires a substantial investment (in terms of man-hours lost, equipment maintenance, materials, etc.) for initial training. On-going training is also required to sustain a satisfactory level of proficiency. There is, therefore, always a need for effective, realistic, and inexpensive complements to conventional schooling to maintain competency. Nowhere is this more true than for the training of steam propulsion engineering plant operators. The complex, almost Gordian knot of valves, piping, and components is overwhelming to the novice and requires continual refresher for qualified watchstanders to perform their jobs effectively and safely. However, the Navy currently has only one computer-

ized steam plant simulator (the Propulsion Plant Trainer (PPT) in Newport, R.I.) and one non-specific stationary hot plant (at Great Lakes Naval Station). "Hands-on" training for prospective division officers and department heads is conducted at one of these two facilities, or on-board ships moored to a pier.

STEAM ENGINEERING TRAINING PROBLEMS

Three problems are evident with the status quo. First, hands-on training focuses on *proper* (i.e., non-catastrophic) operation of the plant. With the exception of the PPT, it is too dangerous to both machinery and human life to impose actual casualty situations on steaming boilers. Therefore, most casualty control training is either learned in the classroom or is simulated. (Simulated casualty control is like kissing one's own sister; it isn't quite the same thing).

The second problem is that training platforms are expensive to maintain. Machinery at the hot plant and onboard ships breaks. The PPT undergoes physical changes to match real-world ship alterations, and these changes often require software updates. Additionally, building a PPT for the West Coast (to fill the training gap) would be a multimillion dollar expenditure. Both the hot plant and "school ships" burn fuel while training. This fuel could be better used getting the ships and their crews underway conducting at-sea operations (where they should be in the first place).

The third problem is that plant line-up changes and casualty restoration is very time consuming. With the exception of the PPT (where restoration is instantaneous), prospective engineering officers spend much of their time on the deckplates answering questions from the instructors and *not* learning by doing. While this problem is non-existent in the PPT, there is only one PPT. The few steam ships stationed in Newport are vir-

tually the only ones that can afford to send watch teams to the trainer.

RESEARCH QUESTIONS

The problems with current on-going fleet steam engineering training form the background for the following questions posed by this paper.

First, can an expert system be developed that effectively and efficiently models boiler operation? If so, can it be designed in such a manner that it can be expanded to model the entire steam plant?

Second, can such a model be constructed using qualitative reasoning such that it is not limited by parameters and features specific to one platform?

Third, must a model-based expert system be written in Lisp or one of the other traditional artificial intelligence languages, or can it be written in a general purpose language such as Ada?

Fourth, can such a system be made inexpensively enough to make it an attractive and affordable shipboard tool?

BoilerModel was developed to answer the questions posed. It is a fairly uncomplicated qualitative model-based reasoning system whose domain is the naval propulsion boiler. It is implemented in Ada. The cause-effect propagation of events in the model-based paradigm is ideally suited for physical applications such as steam generation plants. Model-based systems are beneficial in education and training because they can progress through events causally in much the same manner as students learn. They rely on how components work and how they are interrelated. Thus, plant scenarios can be generated easily by students and abnormal conditions can be diagnosed confidently by watchstanders.

REASONING FROM MODELS

Model-based expert systems have been written in many languages and for many different architectures. Knowledge representation also differs from system to system to suit the specifications of the designers and the needs of the users. However, all of these systems have one thing in common: they reason from some sort of model of the domain. While a rule-based system may reason exclusively from observed values to facts or rules in its knowledge base, model-based systems reason from "first principles," rules which describe the internal processes and causal relationships between components in the domain. Since first principles are facts about objects and how they behave, they can reason from observed values to

real-world states simply by generating different system states, propagating these constraints through the first principles, and comparing the generated sensor values with actual observed values.

"The essence of [the] model-based expert system approach is to generate a model that acts as close to the real world as possible except when a measurement or component fails. . . . When the real world begins to act differently from the model, we detect the discrepancy and diagnose the change using the model."⁴

The heart of the model is constraint propagation. Propagation uses the relationships between components to establish a chain reaction when changes are made to the system. Propagation continues to occur until all valid relationships have been explored. For example, consider the simple valve and piping arrangement in Figure 1 and the corresponding valid set of steam pressure propagation relationships in Table 1.

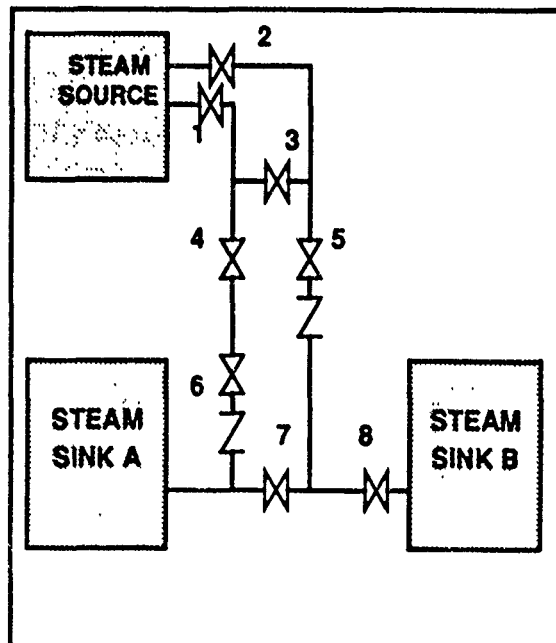


Figure 1

Reasoning about what effect shutting VALVE 1, VALVE 3, and VALVE 7 has on the values of STEAM SINK A and STEAM SINK B would simply be a matter of changing the status of those valves and reevaluating the relationships.

Models themselves fall into two broad groups: quantitative and qualitative. Quantitative models fall outside the scope of this research.

```

VALVE 1 input = NORM
VALVE 1 output = VALVE 1 input (if open) /
                  NONE (if shut)
VALVE 2 input = NORM
VALVE 2 output = VALVE 2 input (if open) /
                  NONE (if shut)
VALVE 3 input = greater of (VALVE 1 output,
                           VALVE 2 output)
VALVE 3 output = VALVE 3 input (if open) /
                  NONE (if shut)
VALVE 4 input = greater of (VALVE 2 output,
                           VALVE 3 output)
VALVE 4 output = VALVE 4 input (if open) /
                  NONE (if shut)
VALVE 5 input = greater of (VALVE 1 output,
                           VALVE 3 output)
VALVE 5 output = VALVE 5 input (if open) /
                  NONE (if shut)
VALVE 6 input = VALVE 4 output
VALVE 6 output = VALVE 6 input (if open) /
                  NONE (if shut)
VALVE 7 input = greater of (VALVE 5 output,
                           VALVE 6 output)
VALVE 7 output = VALVE 7 input (if open) /
                  NONE (if shut)
VALVE 8 input = greater of (VALVE 5 output,
                           VALVE 7 output)
VALVE 8 output = VALVE 8 input (if open) /
                  NONE (if shut)
STEAM SINK A value = greater of (VALVE 6 output,
                                 VALVE 7 output)
STEAM SINK B value = VALVE 8 output

```

Table 1

Qualitative models describe domain components "in terms of causal, compositional or subtypical relationships among objects and events."³ There are several variations of qualitative models. *Classification models* categorize observed patterns to describe processes. The process descriptions identify events which occur over time and in diverse locations. Diagnosing infectious diseases is one example of the use of classification models. *Simulation models* start from a set of initial conditions and predict how the systems will change when the initial conditions are changed. *Functional models* relate system behaviors and states to functional goals.³ White and Frederiksen¹² discuss *phenomenological* and *reductionist models*.

MODEL-BASED vs. RULE-BASED SYSTEMS

Rule-based systems are wholly dependent on facts and rules in their knowledge bases. They cannot, in and of themselves, reason from cause to effect unless the cause and effect happen to be rules accessible to the inference engine. Model-based systems can because cause-effect relationships are easily and naturally modeled as first principles. This is especially important in applications involving physical systems such as steam generation plants and electrical distribution systems. "Rule-based expert systems were never particularly suited to industrial

monitoring applications."⁴ Reasons for this fall into three general areas.

Sensor Failure

Control personnel in real-world industrial systems rely on information from sensors to formulate decisions or perform diagnostics. A rule-based system would require a set of rules mapping possible sensor readings to corresponding plant conditions. A problem arises in that sensor indicators (such as thermometers, pressure gauges, etc.) can themselves fail on occasion. Up to 75% of a rule-based system's knowledge base would consist of rules that could ascertain for any sensor reading whether or not that data is correct.⁵

Model-based systems, on the other hand, have only as many component description and systems interrelationships as are necessary to define the domain. Out-of-limits sensor readings due to faulty sensors can be accurately diagnosed in exactly the same manner as out-of-limits readings due to plant malfunction: components upstream of the sensor in the model are failed in various combinations until a match between model sensor values and real-world sensor values is obtained. If the only match(es) between model and actual system contain contradictory component state information, then the sensor must be faulty (because it is assumed that the real-world system has been accurately and completely modeled).

Number of Rules

The sheer number of rules needed to correctly predict plant performance or diagnose faults in systems of even moderate size is enormous. This plethora of rules presents four problems which are resolved when model-based systems are used. First, as the number of rules/facts increases, the chances of implementing an exhaustive rule base decreases. Since the model-based approach is founded on first principles which describe component behavior and are essentially independent of expert experience, this problem is obviated.

Second, in a large rule-base there may exist some rules which contradict each other, or in concert with each other produce inaccurate results. There may also be rules which are just not correct. A model-based system's network of behaviors, because it focuses first on component or subcomponent behavior and then on relationships, does not grow increasingly more complex as the modeled system grows (although the number of components and inter-component relationships that must be modeled does grow).

Third, a large rule base is expensive in terms of time spent in development. Since such a system would require extensive contact between design personnel and subject matter experts, there would exist a large period of time in which the expert system was in production. Additionally, as the real-world system changes, experts (who do not work for free) would have to be consulted for modifications to the rule base. Although some time lag between conception and implementation would also exist for a model-based system, picking the brains of experts for facts or rules to support all contingencies is unnecessary. Only when new components (which have not been previously modeled) are added will there be a substantial time drain.

Fourth, the addition of new components in rule-based systems increases exponentially the number of new rules needed. Changes to a model-based system would be limited to information about the new component's input and output and values to the components immediately upstream and downstream of it (effectively re-linking the system).

Human Expert

Model-based systems more closely simulate how human experts diagnose faults or predict system behavior. When there is incomplete or conflicting information available, human experts rely on what data is available and formulate hypotheses upon which future actions (e.g., work, casualty control measures, etc.) are based.⁴

Model-based reasoning closely approximates the cause-effect reasoning mechanism employed in human learning. The study of mathematics and science is fraught with facts and figures which are used in problem solving (a cause-effect exercise). The non-quantitative world is also understood analytically. A foreigner unfamiliar with baseball will learn the game more quickly by watching (and doing) than by just memorizing facts and rules.

RELATED WORK

QUALITATIVE PHYSICS FOUNDATIONS

Qualitative model-based reasoning systems have their foundations in the qualitative physics/commonsense reasoning pioneered in the late 1970's and early 1980's by de Kleer, Brown, and Forbus.⁹

ENVISION was developed by de Kleer and Brown at the Xerox Palo Alto Research Center.⁹ It takes a device or component centered view of a system; the system as an entity consists as an integration of many thoroughly specified and described component parts. Device behavior is divided into inter-state and intra-state behaviors and is predicted using the qualitative functions (equations) in the definition of

the device. Prediction is based on propagating known values through the equations of the device, producing both a logical cause/effect link and new facts. Once the intra-state behavior of the device has been determined, all possible future states of that device can be taken from a table that is indexed by the values of state variables and contains all legal states for that device.

Qualitative Process Theory (QPT) was developed in 1982 by Forbus at Massachusetts Institute of Technology.⁹ His *Qualitative Process Engine* uses information about objects and processes to reason about which processes will occur, what they will affect in the system, and when they will stop.⁹ Physical systems are represented as objects which have certain defined interrelationships and processes which are the sole means of changing state in the system. Examples of processes in QPT are heat flow, boiling, and evaporation.

Qualitative simulation (QSIM) was developed by Kuipers in 1985.⁹ QSIM takes a device, functions that describe the behavior of that device, and initial state facts and produces future states into which the device may transition based on the given information. QSIM conducts a breadth-first generation of potential future states, filtering out those that are either redundant or inconsistent with the given facts.

World Qualitative Modeling System (WQMS) was developed in 1990 by Gaglio, Giacomini, Ponassi, and Ruggiero.⁵ WQMS (1) models its domain using Forbus' QPT principles, (2) provides an interface for the user to input values and write results to a file, (3) provides a shell from which various active system views are processed, and (4) uses an Envision (ENV) simulator as well as QSIM simulator to move through the network of possible system states. The difference between the two is that ENV implements a depth-first search while QSIM uses a breadth-first search. Thus, ENV sacrifices the thorough examination of successive states provided by QSIM, but does not get bogged down computationally when used for complex systems. The user is given the option of choosing between the two simulators at the beginning of a session. WQMS was written in the production language OPS5.

MODEL-BASED REASONING IN EDUCATION AND TRAINING

A fundamental problem for students beginning the study of physics or advanced applied mathematics is a lack of conceptualization abilities and an unhealthy reliance on formulaic solutions. Research by White and Frederiksen¹² contends that since traditional teaching relies on the use of

quantitative laws in problem solving, and algebraic reasoning is substituted for underlying causal effects, there is a lack of connection between a student's *instinctive* notions of causality and the quantitative reasoning employed by textbooks and instructors. White and Frederiksen employ the concept of an *articulate microworld* which combines qualitative modeling of electrical circuit behavior within the framework of an intelligent tutoring system.¹² It is the primary vehicle in solving problems where the student is required to formulate mental models to understand domain phenomena and to solve problems. Models of system behavior progress from broadly qualitative and analogous to quantitative based on the student's progress and success in mastering the concepts and system generated test problems of lower level models.

The STEAMER project was initiated in 1979 and developed through 1984 by Hollan in collaboration with several others, principally Hutchins and Weitzman.⁷ The domain of STEAMER is a Navy steam propulsion plant and its goal is to explore the use of artificial intelligence software and hardware in computer aided instruction (CAI). It is written in LISP.

Central to the development of STEAMER is the idea of *mental models*, the models people use to think about complex systems. Graphical interface is very important because the variations of how system interactions are presented are also variations on the level and direction of instruction. STEAMER presents an interactive, inspectable simulation; the user is permitted and encouraged to explore and inspect how system functions perform.

The Intelligent Maintenance Training System was developed by the Behavioral Technology Laboratories at the University of Southern California, funded in part by the Office of Naval Research.¹¹ It is an interactive graphical simulation that allows the user to build a system using a sort of graphical tool box. The user can then specify behavioral rules for each component in the new system. IMTS is implemented in Lisp. IMTS simulations are built from generic objects contained in an object library. *Scenes*, which are screen-sized subsections of the simulation, are built from objects using the *screen editor*. When objects are connected, basic rules regarding the interconnection are automatically generated. Generic objects come pre-coded with behavioral rules indexed by the possible states for the object. Each state has certain conditions which must be true for the state to be true, and certain effects which happen as a result of being in that state.

A Generic Training System (GTS) was developed by Inui, Miyasaka, Kawamura, and Bourne.⁸ Its goal is to effectively use artificial intelligence technology and qualitative reasoning techniques to build an individualized

intelligent tutoring system.⁸ It is written in a variety of languages: Franz Lisp, OPS5, PEARL (Package for Efficient Access to Representations in Lisp) and Flavors. GTS combines knowledge representation schemes used in heuristic (rule-based) systems and qualitative models to offer a more robust training platform than traditional computer aided instruction systems.

GTS is generic enough in principle to be used in a wide variety of intelligent tutoring domains. Since it relies heavily on model-based reasoning concepts, the domain should be one which is adaptive to those concepts. The power distribution prototype that developed as GTS developed has been expanded into a Power Distribution Training System currently in use at the Osaka Gas Training Center.

MODEL-BASED REASONING IN DIAGNOSTICS

Ontological Diagnostic System (ODS), written in LISP in 1989 by Gallanti, Stefanini, and Tomada⁶ relies on knowledge of formal design principles and an understanding of physical laws behind system operation to diagnose malfunctions. Like most model-based systems, the goal of ODS is to provide a deep knowledge network instead of a shallow knowledge base found in rule-based expert systems. However, unlike other model-based systems, ODS does not determine faults by failing likely components, allowing their new values to propagate through the model and then comparing the new model values with the observed system values. Instead, ODS uses models of the faulty behavior of devices to determine faults. The claim of ODS designers is that using these faulty models reduces the complexity of fault diagnosis, thereby making the whole process more effective and practical.⁶ ODS typically performed fault diagnosis in tenths of minutes on a Symbolics 3640 machine with 4 megabytes of main memory.⁶

Hoist is a causal reasoning expert system based on qualitative physics. It was developed by Whitehead and Roach in 1990.¹³ Hoist's domain is fault diagnosis in the lower hoist of the Mark 45 Naval gun turret. It reasons about machine failures from a functional model of the device, and is thus a model-based reasoning system. Hoist is implemented in a language called WIF (What IF), which is based on counterfactual logic. WIF takes a "what if" introduced by the user and assumes it will contradict known facts. The model then generates all known worlds (states) which could exist if the counterfactual clause were true. This is ideal for troubleshooting because instead of matching symptoms to some set of rules, diagnosis starts by introducing suspected fault conditions and ascertaining whether or not the fault state can be reached given the

"truth" of the suspected fault. Hoist runs into combinatorially explosive situations when it is tasked to isolate multiple faults. However, the designers claim that heuristic searches through the "fault space" can reduce the effect of the explosion.¹³

In the late 1980's, Lutch and Zejda developed a fault diagnosis system for chemical processing units based on mathematical models.¹⁰ Lutch and Zejda proposed that all chemical processes, even the most complex, could be broken down into smaller, easier handled subsystems. Since each of the subsystems is described by only a few governing equations, Boolean logic can be used to determine which sensor will fail given actual measurement values in the mathematical models. Problems arise when measurement noise causes the diagnosis to fluctuate between two or more faults. Lutch and Zejda's solution to this problem is to introduce a certain level of belief of failure to each sensor for each discrete level of failure using Shafer-Dempster probability mass distributions.

AN ADA IMPLEMENTATION

From its earliest conception, BoilerModel was meant to be an Ada project. Several factors, including speed, maintainability, and portability contributed to this decision; however, the main consideration was the Department of Defense's embracement of Ada as a *lingua franca* for future programming applications.

SCOPE OF THE MODEL

The original plan for BoilerModel was to write and implement it on an IBM-type PC using Meridian Software's AdaZ (later OpenAda) compiler. An early version of BoilerModel was written and did run with AdaZ; however, the variable stack used by the compiler later proved to be inadequate for the number of global variables (and the size of the data structures in which these variables were instantiated) in the current version of the model. With virtually no changes to existing code, the model was transferred to a Sun SPARCstation and the Verdex Ada compiler. The number and size of global variables did not adversely affect that compiler.

BoilerModel models a somewhat simplified 1200 psi D-type boiler, along with valve and piping systems to and from major loads and supporting auxiliary equipment. Although all propulsion boilers operate the same in principle, BoilerModel's architecture comes from the FF 1052/1078 class platform. For the purpose of this implementation, boiler steam loads are assumed to be "receive-ready" and boiler auxiliaries are assumed to be "supply-ready." This simply means that if, for example, the boiler is on-line and

an open path to the Engineroom for main steam exists, then the steam will be used in the Engineroom (even though, at present, there is no such end-user in the model). Likewise, if there is an open piping path from the Fuel Oil Service Pump, then fuel will flow to the boiler regardless of the fire status of the furnace.

Assumptions like these have their problems. For example, the Main Feed Pumps on a frigate are steam driven. However, since they have not been fully modeled here, they will still operate when steam flow from the boiler is secured. The "receive-ready" and "supply-ready" assumptions should be viewed as temporarily undeveloped components in a larger propulsion plant model. They currently serve as a test harness for the boiler.

The Automatic Boiler Control (ABC) systems were not included in this model; they are complex enough to comprise a separate project. Since they are measurable, interacting physical systems, they can also be implemented in a model-based reasoning system to work with BoilerModel.

Finally, a valve which does not exist on the real-world boiler was included in this model. The Virtual Superheater Outlet was added so the user could observe the effects of stopping all steam flow from the boiler. A later version of BoilerModel should contain a more versatile user interface which would allow the user to change more than one valve status or characteristic per scenario. That versatility is currently lacking.

ADA IN ARTIFICIAL INTELLIGENCE

The typical benchmark in artificial intelligence technology is "adequacy"-- does the system provide acceptably correct answers or diagnoses in an acceptable amount of time or detail? Programs have generally been prototyped in one of the standard AI languages, such as LISP, and once developed, translated into a more efficient language (e.g., C or Pascal).¹

Ada provides an alternate solution. Its rich data types, capability for multitasking, and strong typing requirements are some of the reasons Ada can and should be used from initial program development through implementation of the final product.

Multitasking. In a steam generation plant, several events occur simultaneously. Steam flows through piping systems at the same time as fuel is supplied to the boiler at the same time as feedwater is pumped into the steam drum. Ada tasks are outstanding tools for modeling the cause-effect relationships in such a system. For example, when fires go out in a real-world boiler, steam

flow out of the generation tubes is immediately reduced. Two tasks, one which concerns itself with boiler fires management and another which monitors steam flow through boiler tubes could run independently yet share a common variable: boiler fire status (changeable only by the fires manager). Now, instead of having the disjointed nest of *if* and *case* statements and an unrealistic sequence of events common in a sequential processing system, one can realistically model events which occur concurrently.

Portability and Speed. A machine-dependent artificial intelligence application is useful only as long as the particular machine is available, affordable, and multi-purpose. Similarly, programs written in languages lacking a common standard are neither easily maintained nor readily integrated into other applications written in different dialects of the same language. Lisp and C are languages in which portability can be a problem. C is generally portable, but libraries vary from implementation to implementation. Since C is a language of functions, this can be a difficult problem to overcome.¹ Lisp has traditionally been very nonportable¹, although efforts have been made to standardize Common Lisp. Ada is currently the most portable, "although at present this portability is limited by the availability of Ada compilers and support environments."¹ Since an Ada compiler may only be authorized for use in DoD applications if it conforms to the ANSI/MIL-STD-1815A requirements promulgated by the Department of Defense, it can be a time and money consuming proposition to build a compiler. There are, however, several more on the market since Baker (1987),¹ and they are affordable. Ada's portability was put to the test during the development of BoilerModel. Code for an early version of the project that had compiled and was successfully running on an 80286 machine was transferred in ASCII format to a UNIX based Sun system. No changes to the code were needed for it to compile and run on the new system.

Ada generates code which, while probably somewhat slower than C code, is markedly faster than Lisp. This comes as no surprise; Lisp programs are great consumers of both machine time and memory.¹ One of the design considerations for Ada was real-time control (for use in embedded systems). To that end, one of the three goals established by the Ada language team was *efficiency*. "Any language construct whose implementation was unclear or required excessive machine resources was rejected."² Ada's speed would be of great advantage in real-time expert systems, such as autonomous vehicle control and robot sensor processing.

Readability and Maintainability. An argument can be made that Ada code is easier to read than Lisp code for most people raised on traditional programming.¹ Its En-

glish-like syntax (no *car*'s or *cdr*'s, thank you), minimal use of parentheses, and modular design certainly enhance its appeal. If the language is more readable, then it will probably be more maintainable.² Of course, the bottom line as far as readability goes will probably be personal preference. Ada supporters claim that an Ada program can be understood easily and translated into other languages.¹ In fact, this claim is used to promote the general utility of the language. Can Lisp supporters make such a claim?

ADA vs. LISP -- A CASE-BASED COMPARISON

An early version of BoilerModel (hereafter referred to as ProtoBoiler to differentiate it from the final version) that did not incorporate Ada's tasking constructs was compared to the same program written in Lisp. It should be noted here that the Lisp program was written as functionally as possible to ensure that the comparison fairly evaluated an Ada program against a Lisp program as they are conventionally written. Tables 2, 3, and 4 synopsize the results of the test. The Lisp code was written and run in the Allegro Common Lisp environment in both an uncompiled and a compiled version. The difference in speed is at the expense of storage (16.2 K vice 36.7 K). Since memory is no longer a consideration for all practical purposes, this trade-off is worthwhile. Additionally, both Lisp versions have time for "garbage collection" and "non-garbage collection" use of the CPU. Although garbage collection does vary depending on system usage, it is a real time consumer which must be taken into consideration. The Ada compiler performs garbage collection only once, at compile time. All comparisons were made at the same time of day, with similar system loads.

ADA				
VALVE	TRACE	TIME (SEC)		
		USER	SYSTEM	TOTAL
FOCV	Y	0.3	1.3	1.6
FOCV	N	0.0	0.1	0.1
MSS	Y	0.1	0.3	0.4
MSS	N	0.0	0.0	0.0
FEED STOP	Y	0.0	0.3	0.3
FEED STOP	N	0.0	0.1	0.1
TEST1*	Y	0.7	2.3	3.0
TEST1*	N	0.0	0.2	0.2

Storage: code 15799 bytes
executable 229376 bytes

* TEST1 closes FOCV, then DESUP-IN, then MSS

Table 2

COMPILED LISP						
VALVE	TRACE	NON-GC TIME (SEC)		GC TIME (SEC)		TOTAL
		USER	SYS	USER	SYS	
FOCV	Y	3.7	2.3	0.0	0.0	6.0
FOCV	N	0.2	0.1	0.0	0.0	0.3
MSS	Y	1.1	0.6	0.0	0.0	1.7
MSS	N	0.2	0.0	0.0	0.0	0.2
FEED STOP	Y	1.0	0.6	0.0	0.0	1.6
FEED STOP	N	0.2	0.1	0.0	0.0	0.3
TEST1*	Y	7.5	4.1	0.0	0.0	11.6
TEST1*	N	0.6	0.2	0.0	0.0	0.8

Storage: code 36710 bytes

*TEST1 closes FOCV, then DESUP-IN, then MSS

Table 4

UNCOMPILED LISP						
VALVE	TRACE	NON-GC TIME (SEC)		GC TIME (SEC)		TOTAL
		USER	SYST	USER	SYST	
FOCV	Y	9.3	2.9	0.9	0.9	14.0
FOCV	N	1.0	0.8	0.0	0.0	1.8
MSS	Y	3.0	4.7	0.0	0.0	7.7
MSS	N	0.8	0.7	0.0	0.0	1.5
FEED STOP	Y	2.9	4.6	0.0	0.0	7.5
FEED STOP	N	0.7	0.8	0.0	0.0	1.5
TEST1*	Y	17.6	5.9	1.7	1.1	26.3
TEST1*	N	1.7	0.6	0.0	0.0	2.3

Storage: code 16228 bytes

* TEST1 closes FOCV, then DESUP-IN, then MSS

Table 3

Four test cases were used in the comparison. The first three propagated changes when one valve was closed (Fuel Oil Control Valve in case 1, Main Steam Stop in case 2, and Main Feed Stop in case 3). The fourth case closed three valves (Fuel Oil Control Valve, Desuperheater Inlet, and then Main Steam Stop). The four cases represent the major systems integrated in ProtoBoiler. Each case was run with "trace" on and "trace" off. "Trace" enables the user to watch the propagation of values as they occur. With "trace" off, the user would only see the initial and final plant statuses.

The Ada program ran consistently faster than either Lisp version. "TEST1," which closes multiple valves, ran almost nine times faster than uncompiled Lisp and almost four times faster than the compiled Lisp version (all three with "trace" on). The Ada code required 15.8 K storage versus 16.2 K and 36.7 K for the uncompiled and compiled Lisp versions, respectively. The executable Ada program (which runs independently in the UNIX shell) required 229.4 K. The Lisp code requires the Allegro environment to run. Although, as previously asserted, memory is not a big concern in the test environment (Sun SPARCstation with UNIX operating system), the size of the executable code or all systems required to run the program may be a consideration for other machines. PC's running MS-DOS or PC-DOS may be limited to executable files less than 640 kilobytes.

BOILERMODEL

The prominent data types used in BoilerModel are simple enumeration types, arrays, and records. The main data structure used is the linked list. BoilerModel's two major records, VALVE and BOILER can be found in Figures 2 and 3, respectively. Type MEASUREMENT_VALUE defines the qualitative expressions of actual plant parameters. It can be seen in Figure 4.

```
type VALVE is
  record
    VALVE_ID: INDEX;
    UPSTREAM: VALVE_PTR_ARRAY;
    DOWNSTREAM: VALVE_PTR_ARRAY;
    COUNTED: BOOLEAN:= FALSE;
    NEXT: VALVE_PTR;
    PREV_STATUS: MEASUREMENT_VALUE:= OPEN;
    STATUS: MEASUREMENT_VALUE:= OPEN;
    PREV_INPUT_PRESS: MEASUREMENT_VALUE:=
      NORM;
    INPUT_PRESSURE: MEASUREMENT_VALUE:=
      NORM;
    PREV_INPUT_FLOW: MEASUREMENT_VALUE:=
      NORM;
    PREV_OUTPUT_FLOW: MEASUREMENT_VALUE:=
      NORM;
    INPUT_FLOW: MEASUREMENT_VALUE:= NORM;
    OUTPUT_PRESSURE: MEASUREMENT_VALUE:=
      NORM;
    PREV_OUTPUT_PRESS: MEASUREMENT_VALUE:=
      NORM;
    OUTPUT_FLOW: MEASUREMENT_VALUE:= NORM;
    SYSTEM: SYSTEM_ARRAY_PTR;
  end record;
```

Figure 2

```
type BOILER is
  record
    STEAM_DRUM: DRUM;
    WATER_DRUM: DRUM;
    SUPERHEATER: TUBE;
    DESUPERHEATER: TUBE;
    GENERATION_TUBES: TUBE;
    FURNACE: BOILER_FURNACE;
  end record;
```

Figure 3

```
type MEASUREMENT_VALUE is (NONE, LOW_ALARM,
  LOW, NORM, HIGH, LIFT_SAFE_HI,
  HIGH_ALARM, FUEL_ON_DECK,
  NO_FUEL_ON_DECK, ORANGE,
  BLACK, CLEAR, FOGGED, FAN_SHAPED,
  IRREGULAR, OPEN, SHUT);
```

Figure 4

Type VALVE is a record that contains all the information and parameters necessary for valve operation in BoilerModel. Each valve in the system is an instantiation of an access type, VALVE_PTR, which points to a valve record. The valves in BoilerModel are connected in a linked list structure. Two important fields in type VALVE are UPSTREAM and DOWNSTREAM. UPSTREAM and DOWNSTREAM are arrays of type VALVE_PTR. They contain pointers to the valves which are immediately upstream or downstream of each valve. Normally, there is only one valve in each of these arrays; however, some valves, such as MAIN_STEAM_STOP, act as distributors for several downstream systems or receivers from multiple sources and thus require several valves in one of the two arrays.

Type BOILER is a record of other records. Its constituent members consist of STEAM_DRUM and WATER_DRUM (instances of type DRUM), SUPERHEATER, DESUPERHEATER, and GENERATION_TUBES (instances of type TUBE), and FURNACE (an instance of type BOILER_FURNACE). The fields of the constituent records contain the features and parameters observable in a real-world boiler.

Four tasks drive BoilerModel: PROPAGATE, STEAM_DRUM_MANAGER, FIRES_MANAGER, and TUBE_MANAGER. They are all actor tasks, each embedded in a *loop* statement. Thus, once activated, they continually perform updates and constraint propagation. Since they are all of the same priority, they are scheduled using an implementation-defined First-In, First-Out ready queue.

It should be noted here that the main procedure of any Ada program is an implicit task. Procedure MAIN in BoilerModel is no different. It is assigned a higher priority than any other task so it can perform boiler and plant initialization before propagation begins (unpredictable and erroneous results occurred when MAIN was assigned a

priority equal to the other tasks). MAIN provides user interface by querying the user about what valve status or characteristic to change and displaying boiler status when a boiler parameter has been changed.

Task PROPAGATE takes a look at current and previous status, pressure, and flow for each valve in the linked list of plant valves. If a current and previous parameter of a valve do not match, it must mean that some value was propagated to that valve and must continue propagating until it reaches a sink (user) or a dead end.

Task STEAM_DRUM_MANAGER regulates water level in the boiler steam drum and controls safety valve operation. If there is more flow into the boiler than flow out, water level will increase first to a HIGH condition and then to HIGH_ALARM (signalling a High Water casualty). If the flow out of the boiler is greater than the flow in, water level drops to LOW, and then to LOW_ALARM (a Low Water casualty). NORM water level is the equilibrium condition.

Task FIRES_MANAGER controls steam drum pressure by regulating firing rate based on fuel manifold output pressure. In a real-world boiler, an Automatic Combustion Control (ACC) system regulates fuel and air pressure (and therefore firing rate) to maintain normal steam drum pressure. The ACC system is one part of the Automatic Boiler Controls system which was not incorporated in BoilerModel. However, to mirror real-world boiler operations as closely as possible, boiler firing rate (more accurately, Fuel Oil Control Valve output pressure) changes only when steam drum pressure varies from NORM. When drum pressure is greater than NORM, firing rate decreases; when steam drum pressure drops below NORM, firing rate increases. When steam drum pressure reaches NORM, firing rate becomes NORM. In a real-world steam generation system, NORM depends on the boiler loads (NORM is greater when the ship travels at higher speeds, for example). So, even though the boiler firing rate may not change quantitatively during the transition from abnormal to normal steam drum pressure, it *does* change qualitatively to reflect the new normal fuel and air demand for the changed steam load.

The boiler furnace parameters are also controlled by FIRES_MANAGER via a set of constraints. The constraints constitute the necessary preconditions for furnace values to be other than normal. For example, if there is a path for fuel into the boiler and the fuel is being supplied and fires happen to be extinguished, then the furnace deck will have fuel on it and the periscope will be fogged. How the constraint values come to be is of no concern to FIRES_MANAGER; only the cause-effect relationships across the furnace subcomponents is regulated. Hence,

fuel pressure of NONE to the boiler can result from task PROPAGATE, while a fire appearance of NONE and LOW steam drum pressure are effected by task FIRES_MANAGER.

Task TUBE_MANAGER controls the flow of water and steam through the boiler, from the Manual Check Valve output to the Main and Auxiliary Steam Stops. Figure 5 is a schematic representation of boiler flow. TUBE_MANAGER ensures connectivity by assigning values for component input flow based on the appropriate component output flow.

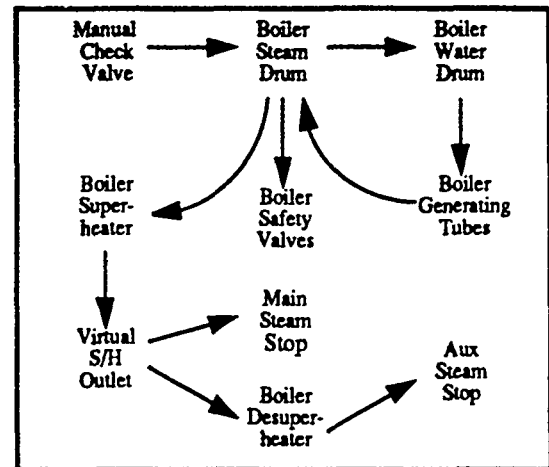


Figure 5

RESULTS

The BoilerModel user interface permits the user to choose between changing a valve *characteristic* or a valve *status*. The characteristics that can be changed are *input flow* and *input pressure*. Each can be changed to *none*, *low*, *norm*, or *high*. Altering a characteristic allows user control over what values will propagate and where propagation will start. A change in valve status more accurately mirrors how an operator can effect changes to the plant: by opening or closing a valve.

In all test cases, the end results of propagation match the expected results in a real-world boiler system. This is a good thing, but could have been accomplished without difficulty using a rule-based expert system. The model-based nature of BoilerModel, however, permits the user to incrementally trace changes as they propagate through the system. Moreover, at any point in time the plant status is *relatively* accurate; events occur and are displayed in correct relation to other events. (e.g., propagation of low steam pressure through the main steam system can occur only after the user can see that something happened to change steam drum pressure).

Consider the effects of closing the Feedwater Control Valve (FWCV) in Figure 6. (The actual runout for this test can be found in the Figure 7 series). Note that the FWCV is the sole entry point for water into the boiler. The first thing BoilerModel does is propagate a NONE value for output pressure and flow downstream of the FWCV. Since there are no branches in the piping system upstream of the valve and the pump is assumed to still be running, a back-pressure is propagated upstream.

At this point the boiler status indicates that there is no flow into the steam drum. The change in flow starts propagating throughout the boiler tubing. Also, since there is a greater flow out of the boiler than there is into it, the steam

drum water level starts to decrease (eventually to the alarm level).

Steam flow is the normal cooling medium for boiler tubes. Since there is now no flow through the generating tubes and boiler fires are still lit, a rupture to those tubes occurs. Also, steam drum pressure has gone down because there is no more water coming into the boiler. This results in a propagation of LOW output pressure and NONE output flow to the Virtual Superheater Outlet (VSH) and on through the steam piping systems.

The superheater also experiences a rupture from a lack of cooling medium. The residual steam in the rup-

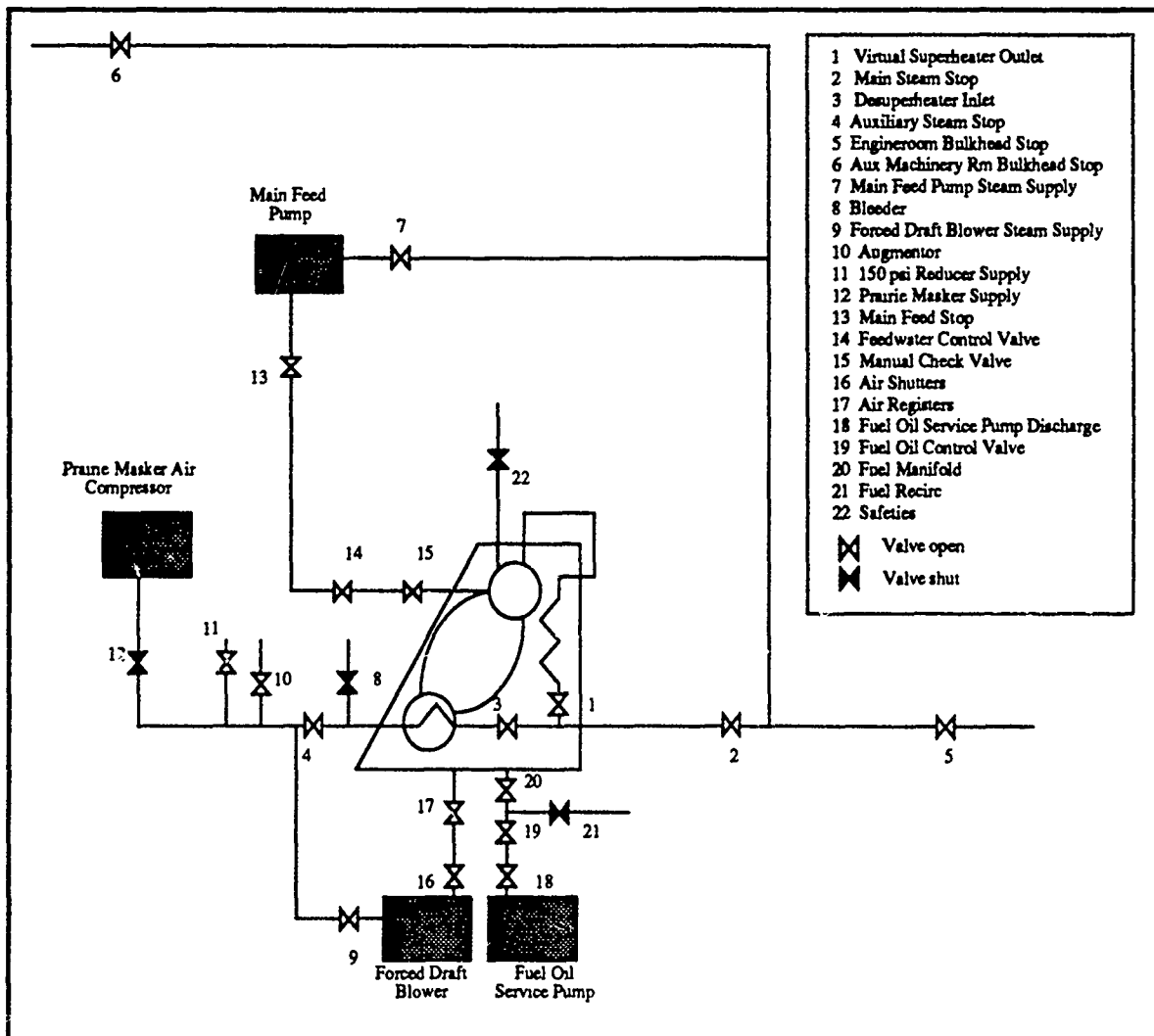


Figure 6

ENTER VALVE TO CHANGE: fwcv

VALVE	STATUS INPUT PRESS		INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
FWCV	SHUT	NORM	NORM	NONE	NONE
VALVE	STATUS INPUT PRESS		INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
FEED_STOP	OPEN	HIGH	NORM	HIGH	NORM
VALVE	STATUS INPUT PRESS		INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
FWCV	SHUT	HIGH	NORM	NONE	NONE
VALVE	STATUS INPUT PRESS		INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
MAN_CHK	OPEN	NONE	NONE	NONE	NONE

BOILER STEAM DRUM WATER LEVEL: NORM BOILER STEAM DRUM PRESSURE: NORM
BOILER STEAM DRUM FLOW IN: NONE BOILER STEAM DRUM FLOW OUT: NONE
BOILER STEAM DRUM TEMP: NORM

BOILER WATER DRUM FLOW IN: NONE BOILER WATER DRUM FLOW OUT: NONE

SUPERHEATER FLOW IN: NORM
SUPERHEATER RUPTURE: FALSE

DESUPERHEATER FLOW IN: NORM	DESUPERHEATER FLOW OUT: NORM
DESUPERHEATER RUPTURE: FALSE	DESUPERHEATER TEMP: NORM

GENERATION TUBES FLOW IN: NONE
GENERATION TUBES RUPTURE: FALSE

FURNACE DECK STATUS: NO_FUEL_ON_DECK FIRING RATE: NORM
BOILER EXPLOSION: FALSE PERISCOPE: CLEAR
FIRES LIT: TRUE FIRE APPEARANCE: FAN SHAPED

////////////////////////////////////

348

PLANT STATUS--BOILER

BOILER STEAM DRUM WATER LEVEL: LOW BOILER STEAM DRUM PRESSURE: NORM
 BOILER STEAM DRUM FLOW IN: NONE BOILER STEAM DRUM FLOW OUT: NONE
 BOILER STEAM DRUM TEMP: NORM

BOILER WATER DRUM FLOW IN: NONE BOILER WATER DRUM FLOW OUT: NONE

SUPERHEATER FLOW IN: NORM SUPERHEATER FLOW OUT: NORM
 SUPERHEATER RUPTURE: FALSE SUPERHEATER TEMP: NORM

DESUPERHEATER FLOW IN: NORM DESUPERHEATER FLOW OUT: NORM
 DESUPERHEATER RUPTURE: FALSE DESUPERHEATER TEMP: NORM

GENERATION TUBES FLOW IN: NONE GENERATION TUBES FLOW OUT: NONE
 GENERATION TUBES RUPTURE: FALSE GENERATION TUBES TEMP: NORM

FURNACE DECK STATUS: NO_FUEL_ON_DECK FIRING RATE: NORM
 BOILER EXPLOSION: FALSE PERISCOPE: CLEAR
 FIRES LIT: TRUE FIRE APPEARANCE: FAN_SHAPED

//

PLANT STATUS--BOILER

BOILER STEAM DRUM WATER LEVEL: LOW_ALARM BOILER STEAM DRUM PRESSURE: NORM
 BOILER STEAM DRUM FLOW IN: NONE BOILER STEAM DRUM FLOW OUT: NONE
 BOILER STEAM DRUM TEMP: NORM

BOILER WATER DRUM FLOW IN: NONE BOILER WATER DRUM FLOW OUT: NONE

SUPERHEATER FLOW IN: NORM SUPERHEATER FLOW OUT: NORM
 SUPERHEATER RUPTURE: FALSE SUPERHEATER TEMP: NORM

DESUPERHEATER FLOW IN: NORM DESUPERHEATER FLOW OUT: NORM
 DESUPERHEATER RUPTURE: FALSE DESUPERHEATER TEMP: NORM

GENERATION TUBES FLOW IN: NONE GENERATION TUBES FLOW OUT: NONE
 GENERATION TUBES RUPTURE: FALSE GENERATION TUBES TEMP: NORM

FURNACE DECK STATUS: NO_FUEL_ON_DECK FIRING RATE: NORM
 BOILER EXPLOSION: FALSE PERISCOPE: CLEAR
 FIRES LIT: TRUE FIRE APPEARANCE: FAN_SHAPED

//

VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
SAFETIES	SHUT	LOW	NONE	NONE	NONE

Figure 7b

PLANT STATUS--BOILER

BOILER STEAM DRUM WATER LEVEL: LOW_ALARM BOILER STEAM DRUM PRESSURE: LOW
 BOILER STEAM DRUM FLOW IN: NONE BOILER STEAM DRUM FLOW OUT: NONE
 BOILER STEAM DRUM TEMP: NORM

BOILER WATER DRUM FLOW IN: NONE BOILER WATER DRUM FLOW OUT: NONE

SUPERHEATER FLOW IN: NONE SUPERHEATER FLOW OUT: NONE
 SUPERHEATER RUPTURE: FALSE SUPERHEATER TEMP: NORM

DESUPERHEATER FLOW IN: NORM DESUPERHEATER FLOW OUT: NORM
 DESUPERHEATER RUPTURE: FALSE DESUPERHEATER TEMP: NORM

GENERATION TUBES FLOW IN: NONE GENERATION TUBES FLOW OUT: NONE
 GENERATION TUBES RUPTURE: TRUE GENERATION TUBES TEMP: HIGH

FURNACE DECK STATUS: NO_FUEL_ON_DECK FIRING RATE: NORM
 BOILER EXPLOSION: FALSE PERISCOPE: FOGGED
 FIRES LIT: TRUE FIRE APPEARANCE: FAN_SHAPED

VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
VSH	OPEN	LOW	NORM	LOW	NORM
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
MSS	OPEN	LOW	NORM	LOW	NORM
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
DESUP_IN	OPEN	LOW	NORM	LOW	NORM
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
BLEED	SHUT	LOW	NORM	NONE	NONE
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
ER_BLKHD_STOP	OPEN	LOW	NORM	LOW	NORM
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
AMR_BLKHD_STOP	OPEN	LOW	NORM	LOW	NORM
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
MFP_STM_SUP	OPEN	LOW	NORM	LOW	NORM
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
ASS	OPEN	LOW	NORM	LOW	NORM
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
AUG	OPEN	LOW	NORM	LOW	NORM

Figure 7c

PLANT STATUS--BOILER

BOILER WATER DRUM FLOW IN: NONE BOILER WATER DRUM FLOW OUT: NONE

DESUPERHEATER FLOW IN: NONE
DESUPERHEATER RUPTURE: FALSE

DESUPERHEATER FLOW OUT: NONE
DESUPERHEATER TEMP: NORM

FURNACE DECK STATUS: NO_FUEL_ON_DECK FIRING RATE: NORM
BOILER EXPLOSION: FALSE PERISCOPE: FOGGED
FIRES LIT: TRUE FIRE APPEARANCE: FAN SHAPED

////////////////////////////////////

Figure 7d

VALVE	STATUS INPUT PRESS		INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
ONE_FIFTY_IN	OPEN	LOW	NONE	LOW	NONE
VALVE	STATUS INPUT PRESS		INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
ER_BLKHD_STOP	OPEN	LOW	NONE	LOW	NONE
VALVE	STATUS INPUT PRESS		INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
AMR_BLKHD_STOP	OPEN	LOW	NONE	LOW	NONE
VALVE	STATUS INPUT PRESS		INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
MFP_STM_SUP	OPEN	LOW	NONE	LOW	NONE
PLANT STATUS--BOILER					
BOILER STEAM DRUM WATER LEVEL: LOW_ALARM BOILER STEAM DRUM PRESSURE: NONE					
BOILER STEAM DRUM FLOW IN: NONE			BOILER STEAM DRUM FLOW OUT: NONE		
BOILER STEAM DRUM TEMP: NORM					
BOILER WATER DRUM FLOW IN: NONE			BOILER WATER DRUM FLOW OUT: NONE		
SUPERHEATER FLOW IN: NONE			SUPERHEATER FLOW OUT: NONE		
SUPERHEATER RUPTURE: TRUE			SUPERHEATER TEMP: HIGH		
DESUPERHEATER FLOW IN: NONE			DESUPERHEATER FLOW OUT: NONE		
DESUPERHEATER RUPTURE: FALSE			DESUPERHEATER TEMP: NORM		
GENERATION TUBES FLOW IN: NONE			GENERATION TUBES FLOW OUT: NONE		
GENERATION TUBES RUPTURE: TRUE			GENERATION TUBES TEMP: HIGH		
FURNACE DECK STATUS: NO_FUEL_ON_DECK FIRING RATE: NORM					
BOILER EXPLOSION: FALSE			PERISCOPE: FOGGED		
FIRES LIT: TRUE			FIRE APPEARANCE: FAN_SHAPED		
////////////////////////////////////					
PLANT STATUS--BOILER					
BOILER STEAM DRUM WATER LEVEL: LOW_ALARM BOILER STEAM DRUM PRESSURE: NONE					
BOILER STEAM DRUM FLOW IN: NONE			BOILER STEAM DRUM FLOW OUT: NONE		
BOILER STEAM DRUM TEMP: NORM					
BOILER WATER DRUM FLOW IN: NONE			BOILER WATER DRUM FLOW OUT: NONE		
SUPERHEATER FLOW IN: NONE			SUPERHEATER FLOW OUT: NONE		
SUPERHEATER RUPTURE: TRUE			SUPERHEATER TEMP: HIGH		
DESUPERHEATER FLOW IN: NONE			DESUPERHEATER FLOW OUT: NONE		
DESUPERHEATER RUPTURE: FALSE			DESUPERHEATER TEMP: HIGH		
GENERATION TUBES FLOW IN: NONE			GENERATION TUBES FLOW OUT: NONE		
GENERATION TUBES RUPTURE: TRUE			GENERATION TUBES TEMP: HIGH		

Figure 7e

FURNACE DECK STATUS: NO_FUEL_ON_DECK FIRING RATE: NORM
 BOILER EXPLOSION: FALSE PERISCOPE: FOGGED
 FIRES LIT: TRUE FIRE APPEARANCE: FAN_SHAPED

VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
VSH	OPEN	NONE	NONE	NONE	NONE
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
MSS	OPEN	NONE	NONE	NONE	NONE
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
DESUP_IN	OPEN	NONE	NONE	NONE	NONE
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
SAFETIES	SHUT	NONE	NONE	NONE	NONE
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
BLEED	SHUT	NONE	NONE	NONE	NONE
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
ER_BLKHD_STOP	OPEN	NONE	NONE	NONE	NONE
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
AMR_BLKHD_STOP	OPEN	NONE	NONE	NONE	NONE
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
MFP_STM_SUP	OPEN	NONE	NONE	NONE	NONE
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
ASS	OPEN	NONE	NONE	NONE	NONE
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
AUG	OPEN	NONE	NONE	NONE	NONE
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
FDB_IN	OPEN	NONE	NONE	NONE	NONE
VALVE	STATUS	INPUT PRESS	INPUT FLOW	OUTPUT PRESS	OUTPUT FLOW
ONE_FIFTY_IN	OPEN	NONE	NONE	NONE	NONE

Figure 7f

ture tubes causes the boiler periscope to fog up. The desuperheater never ruptures because it does not come into direct contact with combustion gases and does not need flow through it to maintain its integrity.

CONCLUSIONS

Model-based reasoning is an effective and efficient method for implementing steam plant engineering training. BoilerModel accurately represents physical components and executes concurrent real-world activities. It provides a usable output that shows how values are propagated through various systems. Answers to the questions posed in the introduction to this paper will be examined, along with other observations/problems encountered.

QUESTIONS ANSWERED

Boiler and Steam Plant Modeling. Expert systems can be developed to efficiently and effectively model a propulsion boiler system. Rule-based systems could conceivably be built to correctly diagnose all casualty situations, but they have several shortcomings. First, they cannot reason beyond the limits of their rule bases. They are, therefore, limited by the knowledge of subject matter experts. Second, the number of rules required for such a system would be enormous because a great deal of them would be required to verify sensor accuracy. Moreover, the large number of rules would slow the expert system down (possibly to the point of uselessness). Third, modifications to the system (including expansion into a steam plant-wide domain) could require substantial alterations to the rule base. Such changes might result in redundant or contradictory rules.

BoilerModel is a streamlined expert system that does not rely on a bank of rules to determine plant status. Instead, it uses cause-effect relations and intra-component behavioral rules to propagate values to their logical conclusions. Since BoilerModel places all emphasis on components and propagation along component connections, modification is simply a matter of modeling new devices and connecting them into the existing system. Therefore expanding BoilerModel into a larger steam plant model is only as difficult as modeling the additional components.

Qualitative Modeling. Designing BoilerModel in the qualitative paradigm posed no problems. In fact, it may have been easier than doing so using actual FF 1052/1078 plant parameters because inexact state descriptions (HIGH, NORM, etc.) require no complicated mathematical formulae. Moreover, qualitative modeling of this project carries two advantages over mathematical or numerical modeling. First, BoilerModel can be used effec-

tively as is by engineering personnel assigned to ships with different types of propulsion boilers and different plant configurations. The general sequence of events that occurs when the feedwater inlet to the boiler is shut is the same for all steam propulsion plants; only the parameters vary. Additionally, because of the component-oriented nature of model-based systems in general and the modularity of BoilerModel in particular, the code can be manipulated to add and remove components or to rearrange valves and piping configurations with very little difficulty. So, although BoilerModel is based on a frigate's steam generation system, it can be modified to match any other steam platform.

The second advantage a qualitative boiler model has over a mathematical one is that the real-world users of the model are plant operators, not mechanical engineers. Although both officer and enlisted watchstanders must know some plant-specific parameters, no one is required to know all of them. One reason is that there are many measurable parameters. Instead of requiring an operator to remember them all (and possibly forget some), engineering guidelines dictate the use of markers (such as red tape) on measuring devices (gauges and thermometers) to indicate the maximum acceptable high or low values. A watchstander can then scan his or her gauge board and observe the relationship of the actual value to the max (or min) for that sensor. In other words, the watchstander makes *qualitative* observations; values are low, high, or normal with respect to the delimiting marker.

Modeling in Ada. Ada proved to be a versatile modeling tool. It provided fairly tight and very fast code. It can be used procedurally or functionally, and is very portable. Lisp code, on the other hand, ran considerably slower than Ada code in the case-based comparison. Moreover, the Lisp code proved more difficult to troubleshoot because it produced run-time errors which, while traceable (using the Lisp "trace" function), were not nearly as easy to locate and correct as compile-time errors in Ada.

Lisp is one of the dominant expert system modeling languages. Should it remain so? To answer yes, a Lisp proponent must provide clear advantages for that language over other contenders. This paper proposes Ada as a language for use in the *full development* of artificial intelligence applications, from prototype to finished product. The only thing Ada lacks is true inheritance in object-oriented programming. That is only temporary; at least one preprocessor, Classic-Ada, allows full use of object-oriented techniques. When tools such as this one are widely available and become a *de facto* part of the DoD standard, then Ada will truly be an *all-purpose* language.

Affordability. Since BoilerModel was developed in Ada, affordability for shipboard use is based on three considerations. First, the initial purchase of an Ada compiler for the PC or Macintosh that can handle the large, numerous global variables inherent in the model. Since Ada is so portable, very little modification to existing code would be required for a changeover from UNIX to MS/PC-DOS or the Macintosh operating system. Second, ships must be equipped with the hardware necessary to run the executable version of BoilerModel. Specifically, PC's or Mac's (preferably laptop versions) need to be accessible to engineering personnel. Third, if this model is to grow any larger than it is, someone needs to make it happen.

The first two considerations involve minimal costs that can easily be reconciled in any budget. The third consideration may involve man-hours (years) diverted toward project development, although some costs can be defrayed by using available research institutions (such as the Naval Postgraduate School).

OTHER OBSERVATIONS AND PROBLEMS

Observations were made and problems encountered during the design and development of BoilerModel that were not directly tied to the research questions.

Compiler Problems. As detailed earlier, BoilerModel's complement of global variables prevented the use of Meridian's standard sized Ada compiler for the PC. This proved to be only a temporary snag; Ada's portability resulted in trouble free transfer to the Verdix Ada compiler on the Suns (although the user-friendly Meridian editing environment was sorely missed).

Incomplete Model. The boiler is probably the single most complicated component to model in the steam plant. There are several valves and subsystems that exist in real-world boiler systems but have not been built into BoilerModel. The reason for this lies with the goal to get a *working* model in Ada completed first. The supporting boiler systems (most notably the Automatic Boiler Control systems) can be added later. To its credit, BoilerModel provides a detailed representation of a propulsion boiler that accurately propagates value changes to their logical conclusions.

Naval Reserve Training. Roughly twelve FF-1052 class frigates are scheduled for reclassification as "FT," or Frigate Trainers. Their function will be to train Naval Reservists on their weekend drills in a non-adversarial environment. Since the typical reservist is not exposed to more than roughly sixteen hours of shipboard duties per month, a portable, computerized trainer could maximize casualty control training while minimizing well-inten-

tioned "mistakes" typical of undertrained deckplate sailors.

OPPE/LOE. All ships must undergo two periodic engineering inspections: the Operational Propulsion Plant Examination (OPPE) and the Light-Off Examination (LOE). Normally, sufficient underway time is allotted for OPPE preparation; however, since part of the exam is materiel readiness, a substantial amount of work at a repair facility is also required. Light-Off Exams are required after extensive yard periods, during which the ship cannot get underway to conduct realistic training. A shipboard training complement like BoilerModel could constructively use the inport time to prepare watchstanders for these inspections.

Other Propulsion Plants. Steam ships are fast becoming an anomaly. With the decommissioning of the Knox class frigates, the Adams and Farragut/Coontz class guided missile destroyers, and the battleships, the only steam driven platforms left will be auxiliaries, cruisers, some amphibious ships, and a handful of aircraft carriers. However, the concepts of model-based design employed in BoilerModel transcend propulsion type and can therefore be applied to both gas turbine and diesel plants.

ABOUT THE AUTHORS

LT James F. Stascavage, USN, has served as Main Propulsion Assistant onboard USS AYLWIN (FF 1081) and as an engineering instructor at the Navy's Steam Engineering Officer of the Watch course and the Propulsion Plant Trainer in Newport, RI. He received an M.S. in Computer Science from the Naval Postgraduate School in September 1991. He can be reached c/o Rt. 2, Box 2602, Mineola, TX 75773.

Dr. Yuh-jeng Lee is currently Assistant Professor of Computer Science at the Naval Postgraduate School. His main research interests are in the areas of automatic programming, automated reasoning, and intelligent systems. He is a member of the Association for Computing Machinery, IEEE Computer Society, and the American Association for Artificial Intelligence. He can be reached by email: ylee@cs.nps.navy.mil.

REFERENCES

1. Baker, L., "Ada and AI Join Forces," *AI Expert*, pp. 39-43, April 1987.
2. Booch, G., *Software Engineering with Ada*, 2d ed., Benjamin/Cummings Publishing Co., 1978.

3. Clancey, W., "Viewing Knowledge Bases As Qualitative Models," *IEEE Expert*, v. 4, pp. 9-23, Summer 1989.

4. Fulton, S. and Pepe, C., "An Introduction to Model-Based Reasoning," *AI Expert*, pp. 48-55, January 1990.

5. Gaglio, S., Giacomini, M., Ponassi, A., and Ruggiero, C., "An OPS5 Implementation of Qualitative Reasoning About Physical Systems," *Applied Artificial Intelligence*, v. 4, pp. 37-65, 1990.

6. Gallanti, M., Stefanini, A., Tomada, L., "ODS: A Diagnostic System Based on Qualitative Modeling Techniques," *1989 IEEE Fifth Conference on Artificial Intelligence Applications*, pp. 141-149.

7. Hollan, J., Hutchins, E., and Weitzman, L., "STEAMER: An Interactive Inspectable Simulation-Based Training System," *The AI Magazine*, pp. 15-27, Summer 1984.

8. Inui, M., Miyasaka, N., Kawamura, K., and Bourne, J., "Development of a Model-Based Intelligent Training System," *Future Generation Computer Systems*, v. 5, pp. 59-69, August 1989.

9. Iwasaki, Y., "Qualitative Physics." In *The Handbook of Artificial Intelligence, Vol. IV*, pp. 323-413. Edited by A. Barr, P. Cohen, and E. Feigenbaum, Addison-Wesley, 1989.

10. Lutch, J. and Zejda, J., "Knowledge Represented by Mathematical Models for Fault Diagnosis in Chemical Processing Units," *Knowledge Based Systems*, v. 3, pp. 32-35, March 1990.

11. Towne, D., Munro, A., Pizzini, Q., Surmon, D., Coller, L., and Wogulis, J., "Model Building Tools for Simulation-Based Training," *Interactive Learning Environments*, v. 1, pp. 33-50, 1990.

12. White, B. and Frederiksen, J., "Causal Models As Intelligent Learning Environments for Science and Engineering Education," *Applied Artificial Intelligence*, v. 3, pp. 83-106, 1989.

13. Whitehead, J. and Roach J., "Hoist: A Second-Generation Expert System Based on Qualitative Physics," *AI Magazine*, v. 11, pp. 108-119, Fall 1990.

Underwater Multi-dimensional Path Planning for the Naval Postgraduate School Autonomous Underwater Vehicle II

Yuh-jeng Lee and J. Bonsignore, Jr
Department of Computer Science
Naval Postgraduate School, Monterey, CA 93943
email ylee@cs.nps.navy.mil

Abstract

Traditionally, path planning software has been developed in LISP or C. Since the recent government mandate for the use of Ada, many researchers are exploring Adas use in a wide variety of areas. This paper seeks to demonstrate the feasibility of using Ada for real-time path replanning. Land vehicle path planning can be accomplished with two horizontal components. For autonomous underwater vehicles, however, the two horizontal components and a vertical component are required to represent three dimensional space. Memory and computational speed restrictions dictate that special processing of the search space be conducted to optimize the time-space trade-off. In this research, a four dimensional array of nodes (two horizontal components, one vertical component and one orientation component) is used to represent the search space. By use of an orientation component, the number of nodes that can be legally moved to is limited, in effect pruning the search space. Search methods implemented were the Tendril search and the Real-time A* search. The Tendril search is a wavefront, breadth-first search. The Real-time A* search uses the Tendril search to a specified search depth and then applies a heuristic to determine the best path to expand upon.

Introduction

Autonomous underwater vehicle (AUV) research continues to grow as more applications are devised. From industry and scientific research to military applications, AUV technology has generated great interest. Currently, there are nearly 30 different organizations researching AUV technology, of which 18 are government funded¹. This indicates the strong interest the government has in this technology.

Due to the AUV's nature, mission planning and execution are very complex problems to solve. Accurate world models must be made and complex path planning performed prior to mission execution. During task performance, continued evaluation of the many aspects of the mission must

be performed. If necessary, adjustment or replanning must be conducted to insure successful mission completion or a decision to abort.

Path Planning: The Tendril Search

General Description

The Tendril search is a wavefront, breadth-first search. Path determination begins by finding the legal moves that can be made from the starting point. These legal moves are saved in a linked list called WAVE and represent the first wave of the propagation. WAVE is subsequently processed one element at a time and generates the next wave in the search process (NEW_WAVE). This process continues until the goal is reached or all legal moves are processed without reaching the goal². Pseudo code for the DO_SEARCH procedure is given:

```
procedure DO_SEARCH is
begin
  read in the search space representation
    from disk
  while the WAVE list is not empty loop
    F_PATH --(Pseudo code listed below)
    exit when the goal is found
  end loop
  print the path
end DO_SEARCH
```

Search Space Representation

Search space representation is achieved by the use of an array or lattice of nodes called N_ARRAY. In effect, this representation parses the search space into a gridded map with each grid unit approximately the size of the AUV (100" by 20" by 10"). Implemented as a record structure, each node contains the STATE, PARENT, and TEND_LEN attributes. These attribute values are determined and used during the planning process. STATE is an integer (either 0 or 1) used to

represent either free or obstacle space. PARENT is an array representing the coordinates of the previous node in the search path. TEND_LEN is the calculated path length at that specific node. The TEND_LEN at the goal is the shortest path length from the starting point to the goal.

Each node in a two dimensional path planning problem has eight neighbors that are legal successors. As the search wave propagates through the search space, each successive wave grows exponentially. Thus, the first wave will have eight legal moves and the second wave 64. Upon expanding the search space to three dimensions, 26 legal moves are possible. As can be imagined, a combinatorial explosion results. To eliminate this problem, a fourth dimension for orientation was incorporated. Similar to the shield representation of obstacles in the configuration space method, each node is associated with a specific orientation³. Since a forward moving vehicle cannot immediately transition to a node at its rear, only nodes in the forward direction were considered as legal successors. This reduced the number of legal moves from 26 to nine, and in effect pruned the search space to a manageable size.

The use of only four orientations, at first, may seem to be a severely limiting factor. This, however, is not the case. Orientations at this level are used for planning purposes and not for the actual navigation of the AUV. A navigation program is used to generate a more refined path for the AUV to use during mission execution. The orientations generated in the Tendril search algorithm are not used in any manner by the navigation process. Instead, a very precise orientation component is calculated in the navigation process⁴. Thus, the use of only four orientations is not a limiting factor and provides adequate means for path generation.

Legal Moves

The legal moves for the node (ROOT) being processed is accomplished with a case statement. Orientation of the node is used to determine the nine successors to be evaluated. Pseudo code for the F_PATH and F_MOVES procedures is given below:

```

procedure F_PATH is
    ROOT : LIST_PTR := WAVE
begin
    while the ROOT is not empty loop
        F_MOVES
        ROOT := ROOT.NEXT

```

```

end loop
if the goal is found then
    return to the DO_SEARCH procedure
end if
WAVE := NEW_WAVE
NEW_WAVE := null
end F_PATH

```

```

procedure F_MOVES (N_ARRAY : in
    out NODE_ARRAY;
    ROOT : in out
    LIST_PTR) is

```

```

    HEADING : integer := the ROOT
    nodes coordinates;

```

```

begin
    case HEADING is
        when the heading is north =>
            CHECK_UP_N;
            CHECK_UP_NE;
            CHECK_UP_NW;
            CHECK_N;
            CHECK_NE;
            CHECK_NW;
            CHECK_DOWN_N;
            CHECK_DOWN_NE;
            CHECK_DOWN_NW;
        when the heading is east => ...
        when the heading is south => ...
        when the heading is west => ...
        when others =>
            null;
        end case;
    end F_MOVES;

```

The procedures called by the case statement (e.g. CHECK_UP_N) calculate the coordinates of the successor for that move and determine the distance from the ROOT to the successor. The CK_STATE and GROW_TEND procedures are used to evaluate the successor node. If it is an obstacle, no further processing for that node is conducted and it is NOT assigned to the NEW_WAVE list. However, if the successor node is free space and has not been previously processed, its TEND_LEN is calculated and the node is assigned to the NEW_WAVE list. Also, if the successor is free space but has been previously processed, it may still be assigned to the NEW_WAVE list if the new TEND_LEN is less than the previously calculated TEND_LEN. Otherwise, no further processing is conducted in this situation.

Program Termination

The program is concluded by printing the path either to a file or to screen. Starting with the goal, its coordinates are printed. By using the coordinates in the PARENT attribute the goals predecessor is printed next. This process continues until the starting point is reached and printed.

Path Replanning: The Real-time A* Search

Introduction

A path replanner is a path planner with more stringent time constraints. It is needed when an AUV is required to circumnavigate an unexpected obstacle to continue its mission. This replanner must, therefore, operate in real-time to facilitate an efficient transition to an alternate path.

The various methods of path planning may not be efficient enough for real-time path planning. As a possible solution, the Real-time A* method was investigated. Many aspects of path planning were considered as well as many questions:

1. What search method should be used to find the frontier node?
2. What should the search depth be?
3. Would the old path be completed disregarded or should a new path try to return to the old path as soon as soon as possible.
4. Should this procedure handle the initial collision avoidance maneuver?
5. How "real" is real-time?

These questions had to be properly answered to produce a true real-time path replanner. Since this thesis predominantly examined the Tendril search.

RTA* Algorithm

The Real-time A* (RTA*) algorithm presented by Korf was modified to incorporate four dimensions⁵. The RTA* can use any search method for path planning, but does so to a specified search depth. In this research the Tendril search method was used in the RTA* algorithm.

The wave of nodes at the search depth is called the frontier and the frontier node with the lowest cost is used for further expansion. All other possible path solutions to that frontier are discarded. By expanding upon the least cost frontier node only, computation time is reduced. The program pseudo-code is provided:

procedure RTA is

```
begin
  GET DATA
  DO SEARCH
end RTA
```

procedure DO_SEARCH is

```
begin
  get the terrain data from file
  while the goal is not found loop
    find the frontier nodes
    pick the node with the estimated least
      cost
  end loop
  print the path
end DO_SEARCH
```

To determine the lowest cost frontier node a heuristic is used. Based upon the sum of the calculated tendril length and an estimate to the goal, the node with the lowest cost is selected for expansion. The estimate to the goal is determined by examining the individual coordinates of the node and comparing them to the components of the goal. The difference between the individual coordinate components is summed together and added to the calculated tendril length. Although not as accurate a method as it could be, it provides adequate estimates for the selection of the frontier node to be expanded.

Conclusions

Even though valid paths are determined using this implementation of the RTA*, the optimal path may not be found. This problem increases as the obstacle density increases. In missions where the obstacle density is very low, the RTA* may be adequate for path replanning needs.

Consideration should be given to reestablishing the AUV on the previously planned path after an obstacle avoidance maneuver. In this situation the path replanner must take into account the previous path and plan a new path to avoid the obstacle yet rejoin the original path as quickly as possible.

No actual experimental data on timing has

been gathered. It is important to understand that the process of recognizing an obstacle, and updating the terrain database will also require some time. The longer this process takes the further out the search depth should be to allow adequate replanning time. Increasing the search depth, however, will require more computation time by the RTA*. Thus, a balance must be found between search depth and computation time.

References:

1. James G. Busby and Joseph R. Vadus, "Autonomous Underwater Vehicle R & D Trends," Sea Technology, Vol. 31, no. 3, pp 66 - 73, May 1990.
2. R. E. McGhee, "Two Dimensional Tendril Search," class notes presented at the Naval Postgraduate School, Monterey, CA, 1990.
3. Tomas Lozano-Perez, "Spatial Planning: A Configuration Space Approach," IEEE, 1983, pp 109 - 119, 1983.
4. Chris Magrino, Three Dimensional Guidance for the NPS Autonomous Underwater Vehicle, Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.
5. Richard E. Korf, "Real-Time Heuristic Search: New Results*," Automated Reasoning, pp 149 - 154.

Dr. Yuh-jeng Lee received his doctorate from the University of Illinois. He currently is a professor of computer science at the Naval Postgraduate School.

Major J. Bonsignore, Jr. received his MSCS from the Naval Postgraduate School in September 1991.

FUTURE DIRECTIONS PANEL

Moderator: Miguel Cario, MTM Engineering, Inc.

Panelists: Jean D. Ichbiah, Original Designer/Ada Language

Paul D. Levy, Rational

Dr. Robert Balzer, USC

John Solomond, Director AJPO

Object Coupling and Object Cohesion In Ada

By Edward V. Berard
Berard Software Engineering, Inc.

PROLOGUE

In Ada, we can implement classes using the following three mechanisms:

- non-generic packages that export a type,
- generic packages that export an object, and
- task types.

We can implement a parameterized class using a generic package that exports a type, a generic package that exports an object, and a generic package that exports a task type. Of course, in all of the above examples, we are assuming that the packages and the task type are created so that they otherwise accurately reflect a class or parameterized class.

Ignoring language tricks (e.g., the inappropriate use of derived types), we can implement inheritance in Ada by hand, i.e., using layers of abstraction. For example, suppose that we have a (generic or non-generic) package that represents a class. If we wish to create a "subclass" of this class, we can create a new package that "withs" the first package and uses the first package as the basis for the creation of the new "subclass." Multiple inheritance can be simulated with the "withing" of several different packages.

OBJECT COUPLING

"[C]oupling is the level to which one module in the system is dependent on other modules. Obviously the greater the amount of coupling between modules, the more complex the design and therefore the harder it will be to understand and maintain."

— [Blair et al, 1991]

"Coupling with regard to modules is still applicable to object-oriented development, but coupling with regard to classes and objects is equally important. However, there is tension between the concepts of coupling and inheritance. On one hand, weakly coupled classes are desirable; on the other hand, inheritance — which tightly couples superclasses and their subclasses — helps us to exploit the commonality among abstractions."

— [Booch, 1991]

In reading about object coupling, one can get the mistaken impression that any object coupling, under any circumstances, is undesirable. Therefore, we need to distinguish between two different categories of object coupling: necessary and unnecessary. Most, if not all, object-oriented applications may be viewed as systems of interacting objects. In such systems it is required (i.e., necessary) that objects be coupled — otherwise no interactions can take place. However, when we design an individual object in isolation, we must minimize the knowledge that this object has about, or requires of, any other object, i.e., the object must be highly decoupled with respect to all other objects.

As a general guideline, the coupling of objects should take place only on an application-by-application basis. Further, even in these situations, care should be taken to minimize the coupling between objects. Finally, we allow for the fact that highly useful collections of interacting (coupled) objects can be created and treated as coherent, cohesive, and useful reusable units (subassemblies).

Unnecessary (premature) coupling of objects should be avoided because:

- Unnecessary object coupling needlessly decreases the reusability of the coupled objects. Specifically, the larger and/or more specialized an object (or system of objects) is, the lower will be the probability that that object (or system of objects) can be reused.
- *Unnecessary object coupling also increases the chances of system corruption when changes are made to one or more of the coupled objects.* Since coupled objects make assumptions about the objects to which they are coupled, changes in these objects can result in unpredictable and undesirable changes in overall system characteristics, e.g., behavior.

[Wild, 1991] identifies two broad categories of object coupling: interface coupling and internal coupling. **Interface coupling** occurs when one object refers to another *specific* object, and the original object makes direct references to one or more items contained in the specific object's public interface. We further stipulate that items other than operations (method selectors), e.g., constants, variables, exportable definitions, and exceptions, may be found in the public interface of an object.

As examples of a fairly loose form of interface coupling, consider three unordered lists, i.e., a list of names, a list of phone numbers, and a list of addresses. We make the following observations regarding these lists:

- Apart from the type of item stored in each list, the implementation of each individual list should be highly consistent with the implementations of the other lists.
- To implement the method for adding an item to a list, we will require that we have access to a method that will "copy" the value of an item to another instance of the same item, i.e., we will need to copy the value of an item into a node in the list. For purposes of this discussion, we are not interested in the details of how the copying occurs (e.g., by passing a pointer or by actually reproducing the values). This "copy method" could, of course, be used by other methods within the list.

In addition, for purposes of this discussion, we will assume that the needed "copy method" is accessible via the public interface of the items being placed in each respective list. For example, the "list of names object" will make use of a copy method encapsulated within the "name object," and made available via a method selector in the public interface.

- Over and above this "copy operation," we will assume that the lists require no additional information about the items they contain.

As we have described it, each list object is coupled (very loosely) to the objects it contains, e.g., the list of names object is coupled to the name objects within it. If, for example, the copy method or its corresponding method selector were to be deleted from the name object, then the list of names object could no longer add name objects to itself. Other, more subtle, changes could also cause problems, e.g., if the name of the copy method selector changed, or if the number or ordering of parameters for the copy method changed.

With the exception of the requirement for an appropriate copy method, each of the list objects makes no assumptions about the objects it contains. You might even say that each list object treats its components as (almost perfectly) black boxes. Yet, as loose as this coupling appears to be, we still have one conceptual problem and one ease of implementation problem.

The conceptual problem involves the separation of the concept of a list from the items contained in a specific list. Specifically, we would like to separate the characteristics of a list from the characteristics that are specific to the items contained in the list. In effect, we would like to identify a set of characteristics that are common to all lists, or, at least, common to lists of names, lists of phone numbers, and lists of addresses in our example.

The ease of implementation problem has two dimensions:

- First, we would like a simple, automatic means of creating new list objects. For example, if we needed a list of computers, we should not have to make a copy of an existing list, and then physically edit that copy to accommodate the necessary changes.
- The second dimension is the specification of the assumptions that the list object makes about its component objects. For example, what specific operations/methods does the list object require from its component objects, and what specific information do these operations/methods require?

The solution to our problem is to have one object (in our example, the list) treat the other objects (in our examples, names, addresses, phone numbers, and computers) as abstractions. Specifically, we will create a "generic list" object, that can, in turn, be instantiated with the necessary information (i.e., the class of objects to be placed in the list, along with an appropriate "copy" operation) to create any desired lists, e.g., lists of names, addresses, phone numbers, or computers. In Ada we accomplish this through the use of a generic package, and require that the class of objects to be placed in the list, along with an appropriate "copy" operation be the instantiation parameters for the generic.

This solution is attractive for several reasons, i.e.:

- It clearly and cleanly separates concepts. In our examples, we can separate the concept of a list from the concepts embodied by the items that can be contained in lists (e.g., names, addresses, phone numbers, and computers).
- It allows one object to *explicitly* state — via parameters — the assumptions that it makes about other objects.

- It simplifies the creation of new categories of objects. Specifically, the instantiation of a template or generic object is usually easier, and far less error prone, than the physical editing of an object.

We refer to this type of object decoupling, i.e., where the assumptions that one object makes about a category of other objects are isolated and used as parameters to instantiate (a template or a generic version of) the original object, as **object abstraction decoupling**.

Now, consider a different example. Imagine a counter object, i.e., an object which is used to count things. The public interface for the counter object contains the following operations: zero (the value of the counter), increment (the current value of the counter), and display (the current value of the counter). The first two operations, i.e., zero and increment, clearly involve only the counter object itself.

The display operation, however, requires access to some form of "output object." In fact, depending on the complexity of the display operation, several other different objects may be involved in displaying the current value of a counter. This operation *tightly* couples the counter object with the output object, and thus, presents a number of problems, i.e.:

- The assumptions that the counter object makes about the output object (and any other objects involved in the displaying process) can only be determined by explicitly examining the source code for the display method. These assumptions can include:
 - the name of the output object,
 - the specific operations in the public interface for the output object that the display method will use to display the value of the counter,
 - the number and forms of the parameters for these specific operations,
 - the intended behavior of these specific operations,
 - the use of any exceptions, constants, variables, and/or other non-operation items contained in the public interface for the output object, and
 - what other objects (besides the output object and the counter object) may be involved in the displaying process, and the assumptions that are made about these objects.

- Any changes made to objects involved in the display method (other than to the counter object itself) may make these objects incompatible with the counter object's display method. The consequences of this very severe. For example, imagine a system where any change to any object may adversely impact the overall system. This means that any time that any object within the system is modified or deleted, the entire system (including the source code for all methods encapsulated in all system objects) must be examined to determine the impact of the change.

- A new system may require the counter object, but may not require that the values of the counter be displayed. A software engineer may elect to create an entirely new counter object rather than modify the existing counter object. If the specific output object (or category of output objects) necessary for the display operation is not present in the new application, reuse of the counter object without changing or deleting the display operation/method is risky.
- When attempting to represent the value of any object to those outside of the object, we must consider the most appropriate form for the representation. For example, will it be textual or numeric, what style will be used (e.g., plain, bold, italic, or underlined), and will the representation take advantage of the media, e.g., one might display information differently depending on the output media — paper, voice, screen, or some other media? The display method must make many assumptions about both the form of the output and the characteristics of the output media. The value will very likely be "displayed" differently depending on the choice of media. Even using the same media (e.g., a color screen), there are many different ways to represent the information.

Decoupling the counter object from the output object (and any other objects necessary for the display method) will not be as simple as decoupling the list object and its components in our earlier examples. The primary reason for this is that while the "add" operation can be viewed as an intrinsic property of a list, the "display" operation is not an intrinsic property of counter objects. The concept of a counter can be sufficiently and completely defined without ever mentioning the fact that the value of a counter may be displayed. Another way of saying this is that while the zero and increment operations are "object-specific," the display operation is "application-specific."

Probably the simplest way to decouple the counter object from the output object (and any other objects involved in the display operation) is to replace the display operation/method with an operation that returns the current value of the counter, e.g., a `value_of` operation/method. In this solution, the counter object is no longer vulnerable to changes in the output object. However, there is a much more important advantage. Software engineers are now free to design applications involving the counter where any of an infinity of things can be done with the counter's values, e.g., they may be used in calculations, stored in databases, or used as components of larger objects.

What we have done specifically is to replace a composite operation/method such as `display`, with a primitive operation, e.g., `value_of`. A **primitive operation/method** is an operation/method that cannot be implemented simply, efficiently, and reliably without knowledge of the underlying implementation of the object in which it is encapsulated. A **composite operation/method** is an operation/method constructed from two or more primitive operations/methods — sometimes from different objects.

We can identify three broad categories of primitive operations, i.e., selectors, constructors, and iterators. The terms "selector" and "constructor" to describe different categories of operations can be traced to the work of Barbara Liskov (e.g., [Liskov and Zilles, 1975]). The concept of an "iterator" had its formal origins in the programming language Alphas (Shaw, 1981), and has been discussed frequently in the literature, e.g., [Cameron, 1989], [Eckart, 1987], [Lamb, 1990], [Ross, 1989], and [Shaw et al, 1981].

Selectors are encapsulated operations which return state information about their encapsulating object, and cannot, by definition, alter the state of the object in which they are encapsulated. (Note that this is a general software engineering definition, and is not to be confused with the concept of a "method selector," e.g., as in Smalltalk.) Selectors are discussed in, e.g., [Bauer and Wossner, 1982], [Booch, 1986], [Booch, 1987], and [Booch, 1991].

The "`value_of`" operation in the counter object is an example of a selector operation. Replacing the `display` operation/method with the `value_of` operation/method is an example of selector decoupling. **Selector decoupling** is the process of replacing an encapsulated composite operation/method with a primitive selector operation with the intended and actual result of decoupling the encapsulating object from other objects.

Next, consider a month object, i.e., an object that represents a month in a Gregorian date (e.g., January ... December). Suppose one of the operations in the public interface for the month object is "`get_month`." When we ask the question "from where or what?", we will probably find that the `get_month` operation couples the month object with at least an "input object," and probably a variety of other objects. This object coupling suffers from all of the problems we mentioned earlier for the `display` operation in the interface to the counter object.

As before, we will replace the composite operation/method (`get_month`) with a primitive operation. However, a selector operation will not be appropriate. We will need a constructor operation. **Constructors** are operations which can (and often do) change the state of their encapsulating object to accomplish their function. We can think of constructors as operations which "construct a new, or altered, version of an object." Constructors are discussed in [Bauer and Wossner, 1982], [Booch, 1986], and [Booch, 1987].

In our example, we will replace the `get_month` operation/method with a "`from_string`" operation/method. The `from_string` operation/method is a constructor. (It will cause a month object to come into existence whose initial value will be derived from a string value.) The `from_string` operation/method effectively decouples the month object from the input object and any other objects that might have been involved in the `get_month` operation/method.

Replacing the `get_month` operation/method with a `from_string` operation/method is an example of constructor decoupling. **Constructor decoupling** is the process of replacing an encapsulated composite operation/method with a primitive constructor operation with the intended and actual result of decoupling the encapsulating object from other objects.

At this point, someone might observe that the `value_of` operation/method in the counter object probably returns an integer object, and the `from_string` operation/method in the month object requires a string object as input. Does this mean that the counter object is coupled to an integer object and that the month object is coupled to a string object? The answer is "no," and the reason is that both integers and strings are primitive objects.

Primitive objects are objects that are:

- *defined in the standard for the implementation language* (note that the standard may encompass more than just the syntax and semantics for the language, e.g., it may include standard libraries of objects and a standard environment), and
- *globally known*, i.e., these are objects that are known (and whose characteristics are known) in any part of any application created using the implementation language.

Please note that primitive objects may be quite complex in nature. We are using the word "primitive" in the "basic building block" sense rather than in the "simplest of all forms" sense.

An object that refers only to itself and to primitive objects is considered, for all intents and purposes, totally decoupled from other objects.

For our next example, consider a "list of names" object. Suppose someone included a "display" operation in the public interface for this list of names object. We would, of course, have all the problems that we previously mentioned for the display operation in the interface for the counter object. However, our problems would be compounded with two additional problems, i.e.:

- The problems of representation form and output media are more complex with objects having composite state than they are with objects having states that can be represented using single monolithic values. For example, will there be multiple columns, will all components be displayed in the same manner, will there be labels and headings, and how will the output be ordered? Displaying a list of names is a much more involved process than displaying an integer representing the current value of a counter.
- Suppose that, instead of displaying all the names in the list, we wish to delete all the names that begin with "N" or change each occurrence of "Hendricks" with "Hendrix"? While each of these operations/methods requires that we iterate over (loop through) the object the specific tasks to be performed at each node in the list vary. It seems that we should be able to "factor out" or separate the iteration capability from the specific tasks we must accomplish.

A **composite object** is an object which is *conceptually* composed of two, or more, other objects. The objects which make up the composite object are referred to as **component objects**. A **heterogeneous composite object** is an object that is *conceptually* composed from objects which are not all *conceptually* the same. A **homogeneous composite object** is a composite object that is *conceptually* composed of component objects which are all *conceptually* the same.

If we are dealing with a homogeneous composite object, we can consider the inclusion of an iterator capability in its interface. An **iterator capability** (often simply referred to as an **iterator**) allows its users to systematically visit all the nodes in a homogeneous composite object and to perform some user-supplied operation at each node. (There is a great deal of technology associated with iterators. For example see Chapter 7 of [Booch, 1987] for a discussion of active (open) iterators versus passive (closed) iterators.)

Returning to our "list of names" object, if we replace the display operation/method with an iterator capability, we will have decoupled the list of names object from the output object. Specifically, the list of names object will offer an iterator capability, and those wishing to display the names on the list will "instantiate" the iterator capability with the operation(s) necessary to display the names contained in the list. This is an example of iterator decoupling. **Iterator decoupling** is the process of replacing an encapsulated composite operation/method with an iterator capability with the intended and actual result of decoupling the encapsulating object from other objects.

Interface coupling occurs when an object references the items in the public interface of another object. Up to this point, our discussion has focused almost exclusively on the coupling related to the *operations* (method selectors) in the public interface. We must also consider any stand-alone constants and variables that may be in the interface. We will add the following items to our discussion:

- Even if our programming language allows for both data and objects, we should avoid the use of data (in the form of either variables or constants) in the public interface for an object. The concept of an object both protects others from changes in the underlying implementation for some information, and embodies (via a set of operations and their corresponding methods) a set of rules regarding the examination and manipulation of that information. Data offers no such protection.

- Except in unusual circumstances, it is far better to have constants in the public interface for an object than it is to have variables. Attempting to determine who changed globally available information, let alone attempting to understand the overall application, is difficult at best. Further, system components that communicate via global information are often difficult to modify and to reuse.
- Even if information is made available in the form of constants in the public interface for an object, it is far better that this information be in the form of discrete scalar items or homogeneous structures, i.e., not in the form of a heterogeneous structure. Objects that have access to these global structures are sensitive to changes in the structure. Further, they may also have access to information that they do not need. (Specifically, they may need only part of the information provided in a heterogeneous item, however, nothing prevents them from accessing other information in the heterogeneous structure.)

We now shift our attention to internal coupling. **Internal object coupling** is present in two situations:

- **Inside internal object coupling** is a normal by-product of object design, and occurs when:
 - the methods for that object are coupled to the encapsulated state information for the object, and/or
 - the component objects that make up a composite object are coupled with the overall composite object.

All objects will exhibit one or both of these forms of inside internal object coupling. However, there are varying degrees of tightness for this form of coupling, and software engineers should strive to keep this variety of coupling as loose as possible.

- **Outside internal object coupling** occurs if an object external to another object has access to, or knowledge of, the underlying implementation of the other object, i.e., that part of an object that we normally consider hidden to those outside of the object.

Internal coupling is much tighter (i.e., much worse) than is interface coupling. While there are indeed some situations where internal object coupling could be justified, these situations should be very rare.

All objects have state. (See, e.g., [Booch, 1991].) State information is often physically stored inside an object. We also allow that some state information can be derived when needed, i.e., it is derived from other physically stored state information.

The internal algorithms (i.e., the methods) by which an object accomplishes its operations must know something about this internally-stored information. Specifically, they must know what information exists, and how to access (and possibly convert) the information they need.

Suppose, for example, that the state information for a given object is stored in the form of a data structure (e.g., an array or a record). All the internal methods for that object can be, and probably are aware of this structure. If this data structure is modified, or if a new data structure is added, the internal methods for the object may have to be modified to accommodate the change.

Good software engineering dictates that we hide (isolate the details of) design/implementation decisions (e.g., [Parnas, 1972], [Parnas, 1979], and [Parnas et al, 1983]). This idea is one of the cornerstones of an object-oriented approach, and it need not be ignored simply because we are inside (as opposed to outside) of an object. For example, [Wirfs-Brock and Wilkerson, 1989] and others have suggested that all access to state information by the internal methods for an object be via "access methods." This approach:

- isolates the internal methods from changes in the form of the internally-stored state information, and
- makes the incremental modification/extension of objects (e.g., the subclassing of methods) easier.

Korson and McGregor ([Korson and McGregor, 1990]) cite the coupling of a composite object with its components. For example, suppose that we have a date object which is, in turn, composed of a month, a day, and a year object. Further suppose that our intention is to have the date object act merely as "a box into which we place a month, a day, and a year," i.e., the date object itself does no verification as to the validity of any given date. Even with this simple form for a date object, we are already making assumptions about the objects involved, e.g. we will have mechanisms for copying the values of day objects, month objects, and year objects into and out of date objects.

To minimize the coupling between date and its component objects, and to more clearly specify the assumptions that the date object makes about its component objects, we can use *object abstraction decoupling* in much the same way we did in our list object example. The chief difference will be that instead of worrying about only one category of object, we will specify three categories of objects as abstractions, i.e., days, months, and years.

Next, suppose that we were interested in a "smart date" object, i.e., one that would not permit date objects with invalid values, e.g., "February 31, 1991." We could still use object abstraction decoupling, but we would have to specify more requirements. For example, we could require that each component object of a date object supply an operation that would return an integer value representing the current value of that component, e.g., "5" for a month whose value was "May."

Notice that this last example points to a conflict between coupling and the complexity of object implementation. To maintain loose coupling, and at the same time create more complex composite objects, requires that our templates (generics) become more complex. Further, more complex interrelationships between a composite object and its component objects may require such things as selector, constructor, or iterator decoupling as well.

Outside internal object coupling is the tightest (worst) form of all object coupling. It has two major forms:

- "coupling from the side" in which an object that is *not* a specialization of another object has access to the underlying implementation of that other object (for example, the first object is not a subclass, a derived class, nor an extension of the other object), and

- "coupling from underneath" in which an object that is a specialization of another object (e.g., the object is a subclass, a derived class, or an extension of the other object) has access to the underlying implementation(s) of one or more of its less specialized predecessors (e.g., its superclasses, base classes, or prototypes).

["From the side" and "from underneath" have their origins in how object relationships are often shown graphically. Specifically, in a top-to-bottom orientation, specializations are most often shown *underneath* their corresponding generalizations. "From the side" implies that an object gains access through an interface other than the specialization interface, i.e., from the side.]

We refer to *outside internal coupling* as the tightest form of coupling because it requires one object to know something about the underlying implementation of another object. This violates one of the most fundamental concepts of object-orientation, i.e., information hiding. Normally, if a software engineer modifies the underlying implementation of an object, but does not alter that object's public interface, and preserves the object's outwardly observable characteristics, then other objects will not have to take these changes into account. Unfortunately, if outside internal object coupling is present, this is not true.

Let us first focus on "coupling from the side." In this form of outside internal coupling, the coupled objects need not have any relationship with each other — other than the fact that they are coupled — yet one object has direct access to the underlying implementation of the other. This type of coupling usually occurs under one of the following conditions:

- The implementation language does *not* directly (syntactically and semantically) support information hiding with respect to objects. This means that there is no effective way to "erect a barrier" between all aspects of the underlying implementation for an object and the outside world. This problem shows up when software engineers attempt to use more conventional (i.e., "non-object-oriented") programming languages (e.g., C, Pascal, and assembly languages) to create object-oriented applications.

We should note, however, that just because a programming language is not considered "truly object-oriented," does not dictate that we must have this problem. For example, Ada's packages and private types ([Ammirati and Gerhardt, 1990], [ARM, 1983], [Bach, 1988], and [Cohen, 1986]) provide the elements necessary to encapsulate and hide the underlying implementation of an object.

- Information hiding for one or more objects has been violated (unintentionally or intentionally) by objects that are not specializations of the objects. This violation may even have occurred through the use of "programming language tricks," i.e., little known, little used, and hard-to-understand aspects of the programming language.
- The programming language is considered "object-oriented," but directly (syntactically and semantically) allows objects to access fully or partially the underlying implementations of other objects. The classic example of this is "friends" in C++. (See, e.g., pages 161-163 in [Stroustrup, 1991].)

We can provide some general guidelines regarding coupling from the side, i.e.:

- Avoid the use of languages that do not directly support (syntactically and semantically) information hiding with respect to objects. Use programming languages that are at least "object-based," i.e., languages that support objects as a language primitive. (See, e.g., [Wegner, 1990].)
- Implement objects in a manner that maximizes and enforces information hiding. When using Ada, use both packages and limited private types, or, at the least, private types.
- Keep the details of the form and structure of the underlying implementation of objects hidden. Users should be able to query or change the state of objects only through encapsulated operations/methods. A change in the form and/or structure of the underlying implementation for an object that does not require a change in the public interface, nor a change in the outwardly observable characteristics for that object, should *not* necessitate a change in other objects in the same system.

- Programming language features that allow objects that are not specializations of an object to access the underlying implementation of that object should be avoided.

Now let's shift our attention to "coupling from underneath." Coupling from underneath is a fairly complex issue that is primarily tied to the syntax and semantics of inheritance. Inheritance directly impacts the strength of the coupling both between an object and its specializations, and between the objects in a specialization hierarchy and other objects.

Inheritance is the means by which an object acquires characteristics from one or more other objects. In this context, we take "characteristics" to mean such things as operations/methods, state information representation mechanisms (e.g., instance variables), exceptions, constants, variables, and any other items that are "inheritable." Markku Sakkinen ([Sakkinen, 1989]) has described two major varieties of inheritance, i.e.:

- "*essential inheritance*" which emphasizes the inheritance of behavior and other outwardly observable characteristics of an object, and
- "*incidental inheritance*" which emphasizes the inheritance of all or part of the underlying implementation of the more general object.

Essential inheritance is more commonly referred to as "inheritance of *specification*," and incidental inheritance is more commonly referred to as "inheritance of *implementation*."

These two views of inheritance correspond to the two most prevalent interpretations as to the purpose of inheritance:

- Some people characterize inheritance as a mechanism for mapping "real world" specialization-generalization hierarchies into software. For example, "vehicle" is a more generalized concept than either "automobile" or "motorcycle," and "military aircraft" is a more specialized concept than is "aircraft." The way that this is accomplished in a given programming language is of secondary interest.
- Others view inheritance primarily as a "code sharing/code reusing" mechanism.

Subtyping is a term that is often used inconsistently with regard to inheritance. For example, Pierre America ([America, 1987]) states that "inheritance is concerned with the implementation of the classes, while the subtyping hierarchy is based on the behaviour of the instances (as seen from the outside, by other objects)." In other words, what Sakkinen and others refer to as "inheritance of specification," America calls "subtyping." On the other hand Alan Snyder ([Snyder, 1986]) describes subtyping as something completely apart from inheritance, i.e., "[subtyping is] the rules by which objects of one type (class) are determined to be acceptable in contexts expecting another type (class)." In fact, [Snyder, 1986] discusses the separation of the "inheritance hierarchy" from the "type hierarchy."

Another dimension of inheritance is single inheritance versus multiple inheritance. In single inheritance an object can acquire characteristics directly from only one other object, e.g., its immediate superclass. In a multiple inheritance scheme, an object can inherit (acquire) characteristics directly from *more than one* object. This sometimes leads to problems. For example, what happens if an object attempts to inherit two or more different characteristics with the same name, each provided by a different parent. All situations that allow for multiple inheritance must also provide some systematic means of resolving such conflicts. As we shall see, multiple inheritance significantly complicates the problems associated with "coupling from underneath."

To fully understand "coupling from underneath," we must realize that objects that are used as templates to create other objects (e.g., classes) have two distinct interfaces:

- an "inheritance interface" that they present only to their specializations (e.g., subclasses, derived classes), and
- a "public interface" to which all other objects (including the specializations of the object) have access.

(See, for example, the discussion in Section 3.3 of [America, 1987].)

We can divide our discussion of coupling from underneath into two areas: an internal form and an external form. The internal form is based on how specializations interact with inherited state representation mechanisms (e.g., instance variables). The external form is concerned with the visibility of inheritance in the public interface of an object. Specifically, we are interested in the degree to which objects that are outside of the inheritance hierarchy for a given object are sensitive to changes in that inheritance hierarchy.

[Wirfs-Brock and Wilkerson, 1989] discussed the potential problems of allowing encapsulated methods for an object to have direct access to the underlying implementations of the encapsulated state information for the same object. [Taenzer et al, 1989], went further, commenting on the problems of inherited state information, i.e., "We have also adopted a coding style of not directly using inherited instance variables, but instead using messaging to access them."

The concept of inheritance alone implies (at least a loose) coupling among the objects within a given inheritance hierarchy. However, the actual implementation mechanisms for inheritance (i.e., the semantics of inheritance) in most object-oriented programming languages can introduce undesirable side effects.

The most obvious problem is the sensitivity to change in the underlying implementation (structure) of inheritable state information. Suppose, for example, that a specialization (e.g., subclass, derived class, or child) knows the structure an inherited instance variable, and takes advantage of (depends on) this structure. Changes in the generalization (e.g., superclass, base class, or parent) that result in changes in the structure of the inheritable instance variable

Alan Snyder, in [Snyder, 1986], cites a number of other problems that can occur as a result of the coupling between an object and the objects that inherit state information from that object, e.g.:

- If an inheriting object can only access inherited state information via operations/methods, and a sufficient (minimum necessary) set of operations/methods for this access are not provided, then the designers of the inheriting object must negotiate with the designers of the object providing the inheritable characteristics for the needed operations/methods.

- Some of the operations that make up a sufficient (minimum necessary) set of operations/methods for the inheritable information may not be appropriate for those who are not specializations of the object providing the inheritable information. In other words, if access to inherited state information is only allowed via operations/methods, we would like the option to provide some or all of these operations/methods via the inheritance interface, and not via the public interface of the object providing the inheritable information. Languages such as Trellis ([Kilian, 1990], [Moss and Kohler, 1987], [O'Brien et al, 1987], and [Schaffert et al, 1986]) allow a software engineer to stipulate that some specific operations/methods are only available to specializations of an object, i.e., the objects that inherit the state information representation mechanisms. (C++ also provides such a mechanism via its "protected" members. See, e.g., section 6.6.1 of [Stroustrup, 1991].)

We have just discussed one aspect of the coupling between an object and those objects that inherit characteristics from that object. Specifically, we have observed that knowledge of the underlying implementation of, or sensitivity to changes in, inheritable state information can result in undesirable and/or unintended side effects. However, this does not mean that we should not allow state information to be inheritable.

Grady Booch has observed ([Booch, 1991]) that "there is tension between the concepts of coupling and inheritance. On one hand, weakly coupled classes are desirable; on the other hand, inheritance — which tightly couples superclasses and their subclasses — helps us to exploit the commonality among abstractions." We know that there are a number of things that we can do to minimize the tightness of this form of internal object coupling, i.e.:

- Whenever possible (and practical), allow access to inherited state information only via messages/operations. Specifically, objects that inherit state information should have little, if any, knowledge of the underlying implementation of this state information.
- If one or more of the operations/methods allowing access to the inherited state information is inconsistent with (does not make sense in) the public interface of the object providing the inheritable state information, seek out mechanisms that will restrict access to these operations/methods to the objects inheriting the information.

- An inheriting object does not necessarily need all possible inheritable state information. Just because an object *can* inherit some information does not mean that it *should* inherit that information. Avoid situations where you have no, or little, control over what state information can be inherited. *Said another way, inheritance should be selective.*

Up to this point, we have discussed "internal object coupling from underneath" from the viewpoint of internal state information representations. However, the coupling between an object and those objects that inherit information from the object can also impact the outside (public interface) of the objects involved in an inheritance relationship. Alan Snyder ([Snyder, 1986]) states the problem in the following manner:

"A deeper issue raised by inheritance is whether or not the use of inheritance itself should be part of the external interface (of the class or the objects). In other words, should the clients of a class (necessarily) be able to tell whether or not a class is defined using inheritance?"

Suppose, for example, that we are working in an environment where inheritance is not selective. This means that anything that has the possibility of being inherited, will be inherited. Imagine that a particular object has 5 separate specializations (e.g., subclasses, derived classes, or child classes). Imagine further that 4 of these specializations require a specific operation/message, but the fifth specialization has no need of that specific operation/message. If we place this operation/message in the original object, then it can be inherited by all 5 of its specializations.

The concept of "types" is often confused with the concept of "classes." A **type** is often defined as "a set of values, and a set of operations applicable to those values." (See, e.g., [IEEE, 1983].) In modern software engineering, types are usually used to dictate which items may participate in the same operation. If we say that a language is **strongly typed**, we mean that, with very few exceptions, only items of the same type may participate in the same operation. For example, we may not be allowed to divide an integer by a floating point number until we first convert the type of the integer value to the proper floating point type, or until we convert the type of the floating point value to the proper integer type. In **weakly typed** (and **untyped**) languages items of different types are allowed to participate in the same operation, even if the result will be nonsensical.

In strongly typed programming languages we often allow software engineers to define types that encompass a subset of the values for a given type. We refer to such types as **subtypes** of the original type. As a general rule, items of the subtype of a specific type may participate in the same operations with items of the original type (i.e., the type from which the subtype was derived). The type-subtype relationship is usually unbounded, i.e., any type, including subtypes, can have a subtype. Typed languages (i.e., languages in which there are two or more types) define, and often allow software engineers to embellish upon, type hierarchies, i.e., relationships among types and their corresponding subtypes.

Software engineers also use the terms "statically typed languages" and "dynamically typed languages." In **statically typed languages**, the type of an item is determined early on (e.g., at compile or link time) and does not change. This allows a software engineer (and, for that matter, the compiler) to determine the legality of both an individual operation and the overall program through a static analysis, i.e., without having to execute the program. **Dynamically typed languages**, on the other hand, allow the type of some items to change during the execution of a program. The type of an item is usually determined based on the context of the operation in which it is participating.

Classes, on the other hand, define structures, e.g., operations/methods, internal state information representation, and exportable constants and exceptions, for objects. Specifically, it is possible to have a programming language that supports classes, but not types, e.g., Smalltalk. It is also possible to have an object-oriented programming language that supports both classes and types, e.g., C++. Eiffel is an example of an object-oriented programming language that is strongly typed.

We can see that it is possible to have a programming language that supports both an inheritance hierarchy and a type hierarchy. Very often the type hierarchy is tightly coupled to the inheritance hierarchy. This can lead to problems. (See, e.g., [Cook et al, 1989], [Madsen et al, 1990], and [Porter, 1992].)

Suppose that we are working with a system in which the typing hierarchy is closely tied to the inheritance hierarchy. Specifically, a situation in which a specialization of a class is also a subtype of the original class. Therefore, if A is a specialization of B, and B is a specialization of C, then, by induction, A is also a subtype of C. Now, further suppose that we are in a somewhat strongly typed system, and that it is important for A to be a subtype of C. If we decide to redesign B so that it is no longer a specialization of C, then instances of A are no longer subtypes of C. This will make previously legal operations involving A illegal. In a very real sense A is closely coupled with C, and is sensitive to changes in both B and C. (This example is closely based on one that appears in [Snyder, 1986].)

One way to prevent such problems is to allow for a clear separation of the inheritance hierarchy and the type hierarchy. Unfortunately, the semantics of most commonly used object-oriented programming languages (e.g., C++) do not easily allow for this, if they allow for it at all.

Multiple inheritance both complicates the previously existing problems, and introduces a few new problems.

In a multiple inheritance scheme, an object that inherits from multiple parents is tightly coupled to these parents. Depending on both the items being inherited, and the conflict resolution mechanism, changes to any of the parents can cause significant changes to the object inheriting the characteristics.

Multiple inheritance is a useful concept and can be used both to accurately reflect the "real world," and to reduce the total amount of source code required for a particular application. However, as we have seen above, there can be problems (and these are not the only problems). When we are designing object-oriented systems (e.g., libraries or applications) we should take care when using inheritance, single or multiple. (See, e.g., [Coggins, 1990].)

Lastly, we should mention that there are a number of metrics available to measure various aspects of object coupling, e.g., [Chidamber and Kemerer, 1991], [Liberherr and Holland, 1989], [Liberherr and Riel, 1988a], [Liberherr and Riel, 1988b], [Liberherr and Riel, 1989], and [Liberherr et al, 1988].

OBJECT COHESION

"Simply stated, cohesion measures the degree of connectivity among the elements of a single module (and for object-oriented design, a single class or object)."

— [Booch, 1991]

"Designs in which the modules (in the case of object-oriented design, objects or classes) exhibit high cohesion are those in which the modules group together parts of the system which are closely related."

— [Blair et al, 1991]

A good deal of the discussion regarding object coupling focuses on relationships among different objects. Object cohesion, on the other hand, is based on the logical and physical relationships that bind an individual object together. The more cohesive an object is, the less susceptible it is to change, i.e., the more stable the object is. The introduction of changes into any individual object usually results in undesirable "ripple effects" (i.e., the propagation of change requirements) in the systems that contain that object. Highly cohesive objects usually require very few, if any, changes.

Object cohesion is an externally discernible concept. Specifically, when we discuss the cohesion of an object, we are *not* referring to its underlying implementation, but rather to the interface it presents to the outside world. The underlying implementation of a given object may indeed be chaotic or incoherent, but this does not affect our assessment of the cohesiveness of that object. (There are other concepts and metrics for dealing with the actual underlying implementation of an object.)

Very little has been written about object cohesion. Even the most detailed presentations do not offer more than two to three pages of discussion on the topic. This is not because the topic is not important, but rather because it is more difficult to describe and quantify. For example, given a Gregorian date (i.e., a date composed of a month, a day, and a year) and an "electronic mail message header" (containing information about the sender, the receiver, the passage of the message through the electronic mail system, and the message itself), it might be "intuitively obvious" that the Gregorian date is more cohesive than the "electronic mail message header," but why?

Timothy Budd ([Budd, 1991]) uses [Yourdon and Constantine, 1979] as the basis for his discussion of object cohesion, but provides very few insights into the topic. Peter Coad ([Coad and Yourdon, 1991]) stipulates that "services" (operations/methods) should be functionally cohesive, there should be no extra "services" or "attributes" (state information), all "services" and "attributes" should be descriptive of the object in which they are encapsulated, and specializations of general concepts should be true specializations — not incoherent extensions.

Korson and McGregor ([Korson and McGregor, 1990]) suggest that encapsulated operations/methods must query or modify state information, inherited characteristics should naturally blend with the additional characteristics in the inheriting object, and that "the ultimate test of cohesion is met by the fact that all these pieces are brought together to represent one concept." [Taylor and Hecht, 1990] advise us that, "Cohesion tells us to make sure all the member variables and methods of a class make sense as part of the class." Finally, Grady Booch [Booch, 1991] observes (borrowing terminology from structured design), "The most desirable form of cohesion is functional cohesion, in which the elements of a class or module all work together to provide some well-bounded behavior."

Because there is so little written down (as opposed to known) about object cohesion, the discussion presented here will be based on my personal experience in guiding the development of over 1,000,000 lines of object-oriented software, experiences of my consulting clients (some of whom have developed object-oriented software systems that exceed 2,000,000 lines of code), a basic knowledge of software engineering, and hints provided by some of the previously mentioned authors. What I am about to present is more than an attempt to qualify and quantify object cohesion. It is also an attempt to make the diagnosis and enhancement of object cohesion a teachable, transferable, and repeatable process.

Why is cohesion in general a more difficult concept to grasp than coupling? The answer is fairly simple. Since coupling requires some form of physical or logical linkage between to items, once we have identified that linkage, we have identified a form of coupling. Removing the linkage removes the coupling, and avoiding the establishment of linkages prevents the establishment of coupling.

If one has a reading knowledge of the syntax and semantics of a given programming language, one can identify many types of linkages. This thinking carries over to textual descriptions and graphical models as well. In fact, regardless of the representation mechanism, if one understands the semantics of that representation mechanism, one can identify (at least some forms of) coupling.

The difficult aspects of coupling are:

- identifying some of the more subtle (less blatant) forms of coupling,
- ranking (ordering) different forms of coupling,
- unambiguously specifying the particular attributes of a particular form of coupling (so as to both know when we have that form of coupling, and to differentiate it from other forms of coupling),
- identifying mechanisms to prevent coupling from occurring, and
- removing coupling once it is in place.

Cohesion, on the other hand, requires that we examine an item in isolation, i.e., apart from any other item, and any application that might use the item. When we hear the phrase "logically-related components," this implies that there is some mechanism for knowing which, if any, of the components are logically related. In addition, we must understand degrees of "logical relatedness," i.e., saying that one item is more cohesive than another item implies that we have some means of assessing how closely related the components of an item are to each other.

The earlier example of the Gregorian date and the "electronic mail message header" gives us a clue. Most of us are much more familiar with dates composed of months, days, and years, than we are with "electronic mail message headers." If we think about it, knowing whether or not a specific type of "electronic mail message header" was cohesive depends on our knowledge and experience with such things. While it would be extremely difficult to reach the age of even 6 years without encountering the concept of a date, many people live their entire lives without having to know about "electronic mail message headers."

In a very real sense, understanding cohesion requires many of the same skills necessary to understand reusability, i.e., to both assess the cohesiveness of an item, and to understand the reusability of that same item, we must have:

- technical knowledge of the application domain(s) in which the item will be used,
- at least some limited amount of experience in constructing, modifying, maintaining, testing, and managing applications in the appropriate application domain(s),
- technical knowledge and experience in the types of items found, created, and modified in the application domain(s), i.e., if we are dealing with object-oriented items, we need to know about, and have experience with, objects, and
- a good technical background in and experience with reusability, and in particular, software reusability.

Those involved with the creation of telephone switching systems will very likely need to know about such items as "trunk groups" and "trouble tickets" to assess the cohesiveness (or lack thereof) in these items. Software engineers developing banking applications will be required to recognize that "annual percentage rates" and "loan amounts" are objects. Embedded systems builders will have to know that items such as switches, lamps, and ports are objects that can be used across a wide variety of embedded applications.

In assessing the cohesiveness of any object (or system of objects), we should be asking questions such as:

- Overall, does the object represent a complete and coherent concept, or does it more closely resemble a partial concept, or a random collection of information? (This will be difficult without the skills mentioned above.)
- Does the object directly correspond to a "real world entity," physical (e.g., a post office, phone number, or insurance policy) or logical (e.g., a queue, a rule, or a unit of time)?

- If the object is a composite object, do all of the component objects directly support the concept of the composite object? For example, if you were to describe the composite object, including a list of its component objects, would a clear majority of the people examining your description agree that all the component objects were necessary components of the composite object?
- If the object is a composite object, are there any unnecessary or highly-application-specific component objects?
- Is the "object" characterized in very non-specific terms, e.g., as a collection of "data," "information," "statistics," or "metrics?"
- Do each of the operations/methods in the public interface for the object perform a single coherent function?
- Do the operations/methods in the public interface represent at least a minimally sufficient set of operations/methods for this object, i.e., can one accomplish all necessary work with the object using only this set of operations/methods? If the answer to this question is "yes," will the answer still be "yes" across a wide variety of applications in which the object can be (re)used?
- Are there any unnecessary, or highly-application-specific operations?
- If the object is really a "system of objects," does the overall system of objects truly represent an object-oriented concept, e.g., as opposed to a functional concept?
- If we are dealing with a system of objects, do all of the component objects directly support, or directly contribute to the support of, the object-oriented concept that the system represents?
- If we are dealing with a system of objects, are there any missing objects?
- If a system of objects presents multiple interfaces to the outside world, do each of the interfaces represent a complete and coherent object-oriented concept, or a coherent, object-oriented view of the system of objects?

- If the object (or system of objects) is removed from the context of the immediate application, does the object, in isolation, still represent a coherent and complete object-oriented concept?

Before we go much further in our discussion, we need to eliminate "non-objects" (sometimes called "pseudo-objects") from our discussion. Most object-oriented (and object-based) programming languages supply a physical encapsulation mechanism, e.g., classes in C++ and Smalltalk, packages in Ada, and modules in Modula-3. Sometimes these encapsulation mechanisms are used to package non-object-oriented concepts, i.e.:

- an "object" containing only functions, i.e., an "object" with no state information. Since these items embody only behavior — and no state information, they are not objects. A common example of this is a "math object," an "object" that contains only mathematical functions. This may be cohesive in a functional sense, but it is not cohesive in an object-oriented sense.
- an "object" containing only data, i.e., an "object" that allows direct access to its encapsulated state information. Since these items do not embody any behavior — only directly accessible constants and variables, they are not objects. A common example of this is the "universal constants object." Again, these items may be cohesive in a data-oriented sense, but they are not at all cohesive in an object-oriented sense.

Even though both of the above can be created using the object-packaging mechanism of our chosen implementation language, we do not consider either of them to be objects, and, hence, they fall outside of our discussion of object cohesion.

We will divide our discussion of object cohesion into two parts. One part will focus on individual objects, the other part will be dedicated to systems of objects.

When we speak of "individual objects" we are referring to objects as they are defined in most common object-oriented programming languages, e.g.:

- classes,
- parameterized classes, and
- instances of the above.

Individual objects are definable using the syntax and semantics of object-oriented (and object-based) programming languages. We include language-definable aggregations of objects (what we have been calling “composite objects”) in our definition of “individual objects.”

In our discussion of individual objects we will first turn our attention to the operations/methods in the public interfaces of these objects. (Virtually all of the arguments we will make apply equally well to inheritance interfaces.)

Earlier we said that a primitive operation/method is an operation/method that cannot be implemented simply, efficiently, and reliably without knowledge of the underlying implementation of the object in which it is encapsulated. We also defined a composite operation/method as an operation/method constructed from two or more primitive operations/methods — sometimes from different objects.

We now extend our discussion to include “sufficient sets of primitive operations/methods” and “complete sets of primitive operations/methods.” For a given object, a **sufficient set of primitive operations/methods** is a minimum set of primitive operations/methods necessary to accomplish all necessary work with the object in which they are encapsulated. Please note that this is not necessarily the set of all primitive operations/methods for the given object, and that, for any given object, there is usually more than one sufficient set of primitive operations/methods.

While a sufficient set of primitive operations/methods allows us to accomplish all necessary work for the object in which they are encapsulated, such sets of operations/methods often suffer from two major problems, i.e.:

- Attempting to accomplish some tasks with only a sufficient set of primitive operations/methods may be awkward and/or difficult.
- A sufficient set of primitive operations/methods may not allow us to fully capture the abstraction represented by the object.

Therefore, we often extend a sufficient set of primitive operations/methods for an object with additional primitive operations/methods. A **complete set of primitive operations/methods** for a given object is that set of primitive operations/methods that both allows us to easily work with the object, and fully captures the abstraction represented by the object. Complete sets of primitive operations/methods are at least equal in size, and are almost always larger, than sufficient sets of primitive operations/methods for the same object.

If we examine the set of operations/methods in the public interface for an object and find that the set of operations/methods in the object’s public interface contains:

- only primitive operations/methods, but does not represent at least a sufficient set of primitive operations/methods for the encapsulating object,
- primitive operations/methods, but there is not a sufficient set of primitive operations/methods, and there are also composite operations/methods present,
- a sufficient set of primitive operations/methods, but it also contains additional composite operations/methods, or
- no primitive operations/methods, i.e., all operations/methods in the object’s public interface are composite operations/methods,

then the object is not as cohesive as it could be. Specifically, well-designed objects should contain only primitive operations/methods in their public interface, and there should be at least a sufficient set of primitive operations/methods— and preferably a well-thought-out complete set of operations/methods.

We should keep the following items in mind:

- Objects that have at least a sufficient set of primitive operations/methods in their public interfaces, but also some composite operations, are typically much more cohesive than objects that do not contain at least a sufficient set of primitive operations/methods in their public interfaces.

- Primitive or composite, all operations/methods in the public interface for a given object must directly support the abstraction represented by the object. Further, the encapsulated operations/methods must make sense even when we consider the object in isolation. Specifically, all encapsulated operations/methods should be application-independent. If the only reason for including an operation/method (primitive or composite) in the public interface for an object is a specific application, then that operation/method should probably be removed from the public interface for the object.

[There are many other issues that could be discussed here. Most of them related to software reusability, software reliability, and efficiency.]

Our discussion of object cohesion in individual objects now shifts to composite objects. A **composite object** is an object that is conceptually composed of two, or more, other objects. (A composite object is said to be an *aggregation* of its component objects.) The objects that make up the composite object are referred to as **component objects**. In addition, the composition of a composite object is externally discernible, i.e.:

- the (externally discernible) state of a composite object is directly affected by the presence or absence of one, or more, component objects, and/or
- those outside of the composite object can directly query or change the states of the component objects via the operations/methods in the public interface of the composite object.

Over and above an assessment of the cohesion of a composite object based on the operations/methods in its public interface, we can judge the cohesion of a composite object based on its externally-discernible component objects. A ranking of the cohesiveness of a composite object, based on its externally-discernible component objects, and ordered roughly in terms of increasing goodness, is:

- the externally-discernible component objects are not related to each other, and, taken as a collection, do not support (or seem to support) a single coherent object-oriented concept, i.e., there is no way to describe the collection other than to list the externally-discernible component objects,

- two, or more, of the externally-discernible component objects appear to have a logical, object-oriented relationship, but the collection of externally-discernible component objects, taken as a whole, does not exhibit such a relationship,
- although the collection of externally-discernible component objects, taken in isolation, does not represent a single stable object-oriented concept, the externally-discernible component objects are bound together by how they are used in a particular application, or set of applications, e.g., they are all part of the information displayed on a single screen,
- a definite majority of the externally-discernible component objects are necessary to support a single, coherent, object-oriented concept, but, when the composite object is considered in isolation, there is at least one externally-discernible component object that does not directly support the single, coherent, object-oriented concept (there is an excellent chance that these extraneous externally-discernible component objects were included for a specific application or set of applications),
- all of the externally-discernible component objects are necessary to support a single coherent object-oriented concept, but even though a definite majority of the necessary externally-discernible component objects are present, there are one, or more, externally-discernible component objects that are missing, and
- all of the externally-discernible component objects necessary to support a single coherent, application-independent, object-oriented concept are present, and there are no additional externally-discernible component objects.

CONCLUSION

We have discussed and reviewed the foundations for coupling and cohesion, and examined each in an object-oriented context. While many of the original (non-object-oriented) concepts do carry over into object-oriented software engineering, some have to be enhanced, and new ones had to be generated.

It is unfortunate that there is much more written about object coupling than there is about object cohesion, this is most probably because cohesion does not lend itself to easily identifiable characteristics in the same manner as coupling. You might say that coupling is more of a physical phenomenon and cohesion is more of a logical phenomenon.

Although we have presented a more detailed view of object cohesion than has previously been discussed, much work remains to be done. For example, a nomenclature scheme needs to be developed for the varying types of object cohesion we have described above.

BIBLIOGRAPHY

- [Ammirati and Gerhardt, 1990]. J. Ammirati and M. Gerhardt, "Using Object-Oriented Thinking to Teach Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 25-28, 1990, pp. 277 - 300.
- [America, 1987]. P. America, "Inheritance and Subtyping In a Parallel Object-Oriented Language," *ECCOP '87: Proceedings of the European Conference on Object-Oriented Programming, Lecture Notes on Computer Science, Volume 276*, Springer Verlag, New York, New York, 1987. pp. 234 - 242.
- [ARM, 1983]. Reference Manual for the Ada Programming Language, *ANSI/MIL-STD 1815A (1983)*, United States Department of Defense, February 1983.
- [Bach, 1988]. W.W. Bach, "Is Ada Really an Object-Oriented Programming Language," *Proceedings of Ada Expo 1988*, Galaxy Productions, Frederick, Maryland, 1988, 7 pages.
- [Bauer and Wossner, 1982]. F.L. Bauer and H. Wossner, *Algorithmic Language and Program Development*, Springer-Verlag, New York, New York, 1982.
- [Blair et al, 1991]. G. Blair, J. Gallagher, D. Hutchison, and D. Sheperd, *Object-Oriented Languages, Systems and Applications*, Halsted Press, New York, New York, 1991.
- [Booch, 1986]. G. Booch, "Object Oriented Development," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 211 - 221.
- [Booch, 1987]. G. Booch, *Software Components With Ada*, Benjamin/Cummings, Menlo Park, California, 1987.
- [Booch, 1991]. G. Booch, *Object-Oriented Design With Applications*, Benjamin/Cummings, Menlo Park, California, 1991.
- [Borning, 1986]. A.H. Borning, "Class Versus Prototypes in Object-Oriented Languages," *Proceedings of the 1986 Fall Joint Computer Conference*, IEEE Catalog Number 86CH2345-7, IEEE Computer Society Press, Washington, D.C., 1986, pp 36 - 40.
- [Budd, 1991]. T. Budd, *An Introduction to Object-Oriented Programming*, Addison-Wesley, Reading, Massachusetts, 1991.
- [Cameron, 1989]. R.D. Cameron, "Efficient High-Level Iteration With Accumulators," *ACM Transactions on Programming Language Systems*, Vol. 11, No. 2, April 1989, pp. 194 - 211.
- [Chidamber and Kemerer, 1991]. S.R. Chidamber and C.F. Kemerer, "Towards a Metrics Suite for Object-Oriented Design," *OOPSLA '91 Conference Proceedings, Special Issue of SIGPLAN Notices*, Vol. 26, No. 11, November 1991, pp. 197 - 211.
- [Coad and Yourdon, 1990]. P. Coad and E. Yourdon, *OOA — Object-Oriented Analysis, 2nd Edition*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.
- [Coad and Yourdon, 1991]. P. Coad and E. Yourdon, *Object-Oriented Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [Cohen, 1986]. N.H. Cohen, *Ada As a Second Language*, McGraw-Hill, New York, New York, 1986.
- [Coggins, 1990]. J.M. Coggins, "Designing C++ Class Libraries," *Proceedings of the C++ Conference, San Francisco, California, April 1990*, USENIX Association, Berkeley, California, 1990, pp. 25 - 35.
- [Cook et al, 1989]. W. Cook, W. Hill, and P. Canning, "Inheritance Is Not Subtyping," Report STL-89-17 (Revision 1), Hewlett-Packard Laboratories, Palo Alto, California, 1989, 11 pages. Also in the *Proceedings of the Seventeenth Symposium on Principles of Programming Languages*, January 1990, pp. 125 - 135.
- [Courtois, 1985]. P.J. Courtois, "On Time and Space Decomposition of Complex Structures," *Communications of the ACM*, Vol. 28, No. 6, June 1985, pp. 590 - 603.

- [Eckart, 1987]. J.D. Eckart, "Iteration and Abstract Data Types," *SIGPLAN Notices*, Vol. 22, No. 4, April 1987, pp. 103 - 110.
- [Ejiogu, 1991]. L.O. Ejiogu, *Software Engineering With Formal Metrics*, QED Technical Publishing Group, Boston, Massachusetts, 1991.
- [IEEE, 1983]. IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, The Institute of Electrical and Electronic Engineers, New York, New York, 1983.
- [Kilian, 1990]. M. Kilian, "Trellis: Turning Designs Into Programs," *Communications of the ACM*, Vol. 33, No. 9, September 1990, pp. 65 - 67.
- [Korson and McGregor, 1990]. T. Korson and J.D. McGregor, "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, Vol. 33, No. 9, September 1990, pp. 40 - 60.
- [Lamb, 1990]. D.A. Lamb, "Specification of Iterators," *IEEE Transactions on Software Engineering*, Vol. 16, No. 12, December 1990, pp. 1352 - 1360.
- [Liberherr and Holland, 1989]. K.J. Liberherr and I.M. Holland, "Assuring Good Style for Object-Oriented Programs," *IEEE Software*, Vol. 6, No. 5, September 1989, pp. 38 - 48.
- [Liberherr and Riel, 1988a]. K.J. Liberherr and A.J. Riel, "Demeter: A Case Study of Software Growth Through Parameterized Classes," *Proceedings of the 10th International Conference on Software Engineering*, April 11-15, 1988, pp. 254 - 264.
- [Liberherr and Riel, 1988b]. K.J. Liberherr and A.J. Riel, "Demeter: a CASE Study of Software Growth Through Parameterized Classes," *Journal of Object-Oriented Programming*, Vol. 1, No. 3, August/September 1988, pp. 8 - 22.
- [Liberherr and Riel, 1989]. K.J. Liberherr and A.J. Riel, "Contributions to Teaching Object-Oriented Design and Programming," *OOPSLA '89 Conference Proceedings, Special Issue of SIGPLAN Notices*, Vol. 24, No. 10, October 1989, pp. 11 - 22.
- [Liberherr et al, 1988]. K.J. Liberherr, I. Holland, and A.J. Riel, "Object-Oriented Programming: An Objective Sense of Style," *OOPSLA '88 Conference Proceedings, Special Issue of SIGPLAN Notices*, Vol. 23, No. 11, November 1988, pp. 323 - 334.
- [Liskov and Zilles, 1975]. B. Liskov and S.N. Zilles, "Specification Techniques for Data Abstraction," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, March 1975, pp. 7 - 19.
- [Liskov et al, 1977]. B.H. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, Vol. 20, No. 8, August 1977, pp. 564 - 576.
- [Liskov et al, 1981]. B.H. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder, *CLU Reference Manual*, Springer-Verlag, New York, New York, 1981.
- [Madsen et al, 1990]. O.L. Madsen, B. Magnusson, and B. Møller-Pedersen, "Strong Typing of Object-Oriented Languages Revisited," *OOPSLA/ECOOP '90 Conference Proceedings, Special Issue of SIGPLAN Notices*, Vol. 25, No. 10, October 1990, pp. 140 - 150.
- [Moss and Kohler, 1987]. J.E.B. Moss and W.H. Kohler, "Concurrency Features for the Trellis/Owl Language," *Proceedings of the European Conference on Object-Oriented Programming 1987*, Paris, France, pp. 223 - 232.
- [O'Brien et al, 1987]. P.D. O'Brien, D.C. Halbert, and M.F. Kilian, "The Trellis Programming Environment," *OOPSLA '87 Conference Proceedings, Special Issue of SIGPLAN Notices*, Vol. 22, No. 12, December 1987, pp. 91 - 102.
- [Parnas, 1972]. D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems Into Modules," *Communications of the ACM*, Vol. 5, No. 12, December 1972, pp. 1053-1058.
- [Parnas, 1979]. D.L. Parnas, "Designing Software for Ease of Use and Extension," *IEEE Transactions on Software Engineering*, Vol. 5, No. 2, March 1979, pp. 128 - 157.
- [Parnas et al, 1983]. D.L. Parnas, P.C. Clements, and D. M. Weiss, "Enhancing Reusability with Information Hiding," *ITT Proceedings of the Workshop on Reusability in Programming*, 1983, pp. 240 - 247.
- [Porter, 1992]. H.H. Porter, III, "Separating the Subtype Hierarchy from the Inheritance Implementation," *Journal of Object-Oriented Programming*, Vol. 4, No. 9, February 1992, pp. 20 - 22, 24 - 29.

- [Ross, 1989]. D. Ross, "The Form of a Passive Iterator," *Ada Letters*, Vol. 9, No. 2, March/April 1989, pp. 102 - 105.
- [Sakkinen, 1989]. M. Sakkinen, "Disciplined Inheritance," *ECOOP '89: Proceedings of the European Conference on Object-Oriented Programming, British Computer Society Workshop Series*, Cambridge University Press, Cambridge, United Kingdom, 1989, pp. 39 - 56.
- [Schaffert et al, 1986]. C. Schaffert, T. Cooper, B. Bullis, M. Killian, and C. Wilpolt, "An Introduction to Trellis/Owl," *OOPSLA '86 Conference Proceedings, Special Issue of SIGPLAN Notices*, Vol. 21, No. 11, November 1986, pp. 9 - 16.
- [Shaw, 1981]. M. Shaw, Editor, *Alphard: Form and Content*, Springer-Verlag, New York, New York, 1981.
- [Shaw et al, 1981]. M. Shaw, W.A. Wolf, and R. London, "Abstraction and Verification in Alphard: Iteration and Generators," *Alphard: Form and Content*, Springer-Verlag, New York, New York, 1981, pp. 73 - 116.
- [Snyder, 1986]. A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *OOPSLA '86 Conference Proceedings, Special Issue of SIGPLAN Notices*, Vol. 21, No. 11, November 1986, pp. 38 - 45.
- [Snyder, 1987a]. A. Snyder, "Inheritance and the Development of Encapsulated Software Components," *Proceedings of the Twentieth Hawaii International Conference on System Sciences, Kona, Hawaii*, January 1987, pp. 227 - 233.
- [Snyder, 1987b]. A. Snyder, "Inheritance and the Development of Encapsulated Software Components," *Research Directions in Object-Oriented Programming*, The MIT Press, Cambridge, Massachusetts, 1987, pp. 165 - 188.
- [Stroustrup, 1991]. B. Stroustrup, *The C++ Programming Language, Second Edition*, Addison-Wesley, Reading, Massachusetts, 1991.
- [Taenzer et al, 1989]. D. Taenzer, M. Ganti, and S. Podar, "Problems in Object-Oriented Software Reuse," *ECOOP '89: Proceedings of the European Conference on Object-Oriented Programming, British Computer Society Workshop Series*, Cambridge University Press, Cambridge, United Kingdom, 1989, pp. 25 - 38.
- [Taylor and Hecht, 1990]. D.K. Taylor and A. Hecht, "Using CASE for Object-Oriented Design with C++," *Computer Language*, Vol. 7, No. 11, November 1990, pp. 49 - 57.
- [Wegner, 1990]. P. Wegner, "Concepts and Paradigms of Object-Oriented Programming," *OOPS Messenger*, Vol. 1, No. 1, August 1990, pp. 7 - 87.
- [Wild, 1991]. F.H. Wild III, "Managing Class Coupling: Apply the Principles of Structured Design to Object-Oriented Programming," *UNIX Review*, Vol. 9, No. 10, October 1991, pp. 44 - 47.
- [Wirfs-Brock and Wilkerson, 1989]. A. Wirfs-Brock and B. Wilkerson, "Variables Limit Reusability," *Journal of Object-Oriented Programming*, Vol. 2, No. 1, May/June 1989, pp. 34 - 40.
- [Wirfs-Brock et al, 1990]. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.
- [Yourdon and Constantine, 1979]. E. Yourdon and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

AUTHOR'S INDEX

Name	Page	Name	Page
Abbott , Russell J	34	Joiner, Harry	220
Agrawal, Jagdish	74	Labhart, J	319
Al-Dhelaan, Abdullah	74	Lander, Leslie C	285
Allers, Hilary	55	Latour, Larry	247
Bates, Paul D	116	Leach, Ronald J	68
Bender, Mary	294	Lee, Yuh-Jeng	337, 357
Berard, Edward V		Macpherson, George W	305
Black, Harlan	134	Martin, Dennis S	42
Bonsignore, J. Jr	357	Meadow, Curtis	247
Brown, Russ	319	Meier, M	202
Burnham, C. Alan	332	Menell, Ray	139
Cannella, John K	24	Mitra, Sandeep	285
Cheng, M	147	Morell, Larry	98
Coleman, Don M	68	Morgan, Judy	319
Corbett, Lindon J	125	Naiditch, David	28
Cowderoy, A J C	323	O'Connor, Michael J	110
Dobbs, Verlynda	332	Racine, Glenn	192
England, Jeffrey E	174	Richards, P.M.	233
Fenick, Stewart	220	Ritchie, Roger	174
Fowler, Joyce	208	Rondogiannis, P	147
Gaumer, Dale	202	Salih, Sabah	46
Gerlich, Rainer	276	Scandura, Joseph M	167
Gilroy, Kathleen	258	Skinner, John	46
Goel, Arvind	294	Solderitsch, James	15
Heidelberg, L	139	Stascavage, James F	337
Herleth, Bill	208	Tan, Lu-Ping	68
Hobbs, R.	192	Vandersluis, Kirstan	233
Hollingsworth, Joe	82	Vasilescu, Eugen N	46
Hoolihan, Joseph P	125	Weide, Bruce W	82
Hooper, James W	110	Willis, Robert A. Jr	98
Jeffcoat, Bart	116	Wright, Elena	11
Jenkins, J. O	323	Zage, Wayne M	202
Johnson, Patrice	208	Zage, Dolores M	202